SAPHIRE TECHNICAL REFERENCE

Systems Analysis Program for Hands-on Integrated Reliability Evaluations (SAPHIRE) Version 6.0

Technical Reference Manual

Idaho National Engineering Laboratory
Lockheed Martin Idaho Technologies Company, Inc.
Idaho Falls, ID 83415

Prepared for the
Division of Systems Technology
Office of Nuclear Regulatory Research
U. S. Nuclear Regulatory Commission
Washington, D.C. 20555

Abstract

The Systems Analysis Programs for Hands-on Integrated Reliability Evaluations (SAPHIRE) refers to a set of microcomputer programs that were developed to create and analyze probabilistic risk assessments (PRAs), primarily for nuclear power plants. Herein information is provided on the principles used in the construction and operation of Version 6.0 of the SAPHIRE system. It summarizes the fundamental mathematical concepts of sets and logic, fault trees, and probability. This volume then describes the algorithms that these programs use to construct a fault tree and to obtain the minimal cut sets. It gives the formulas used to obtain the probability of the top event from the minimal cut sets, and the formulas for probabilities that are appropriate under various assumptions concerning reparability and mission time. It defines the measures of basic event importance that this program can calculate. This volume gives an overview of uncertainty analysis using simple Monte Carlo sampling or Latin Hypercube sampling, and states the algorithms used by this program to generate random basic event probabilities from various distributions.

Executive Summary

The Systems Analysis Programs for Hands-on Integrated Reliability Evaluations (SAPHIRE) refers to a set of microcomputer programs that were developed to create and analyze probabilistic risk assessments (PRAs), primarily for nuclear power plants. Herein information is provided on the principles used in the construction and operation of Version 6.0 of the SAPHIRE system. Set theoretic operations and relations are summarized, and their relation to Boolean logic is explained. Fault trees are reviewed, including all of the gate types allowed by SAPHIRE. Finally, the rules of probability are summarized.

The procedure by which SAPHIRE builds a fault tree from the user inputs, simplifies and truncates it according to the user's specifications, and determines the minimal cut sets is outlined. SAPHIRE is written in a recursive language, and performs many operations by recursive procedures. It initially takes the user's input and builds a simplified internal representation of the tree. This involves several steps:

- linking portions that were connected by transfer gates,
- expanding N/M gates as combinations of OR and AND gates,
- determining the unique TOP gate,
- checking for logical loops,
- pruning portions of the tree having house events, and
- coalescing like gates.

To obtain the minimal cut sets in an efficient way, SAPHIRE searches for independent subtrees and for modules, both of which are treated as single tokens until very late in the process. It then determines the optimal order for processing the tree, based on the levels of the gates, and begins making a list of cut sets. Based on the basic event probabilities or sizes (and the user's truncation specifications), it eliminates some cut sets early in the process. It also eliminates nonminimal cut sets, those that can be absorbed by other simpler cut sets, and finally obtains a list of minimal cut sets that the user has specified should not be truncated. The last step is to combine the fault trees for failures of different systems, to obtain the fault tree for an accident sequence involving the failure of certain systems and the success of others.

Selected formulas are provided:.

formula for the probability of a cut set,

approximations for the probability of a union of cut sets,

formula for the frequency of an accident sequence

formulas for reliability and unavailability of repairable and nonrepairable components, corresponding to the probabilities of various basic events, formulas for different measures of importance of a basic event.

Uncertainty analyses are performed by Monte Carlo simulation, with the basic event probabilities drawn from user-specified distributions. Two types of simulation are possible in SAPHIRE, simple Monte Carlo sampling and Latin Hypercube sampling. The sampling distributions that are supported by SAPHIRE are presented, and the algorithms used for generating random numbers from these distributions are documented. Correlation classes, allowing the user to state that certain basic event probabilities are equal although both are uncertain, are also explained. A simple example illustrates the two types of simulation.

Also included is an example showing how SAPHIRE finds the minimal cut sets of a fairly simple fault tree, and how SAPHIRE finds the probability of the TOP event and the importance of the basic events.

Foreword

The U.S. Nuclear Regulatory Commission has developed a powerful suite of personal computer programs for the performance of probabilistic risk assessments (PRAs). This system, known as the Systems Analysis Programs for Hands-on Integrated Reliability Evaluations (SAPHIRE), allows an analyst to perform many of the functions necessary to create, quantify, and evaluate the risk associated with a facility or process being analyzed. This program includes software to define the database structure, to create, analyze, and quantify the data, and to display results and perform sensitivity analyses. This system provides the functionality for taking a PRA from the conceptual state all the way to publication.

SAPHIRE is a program developed for the purpose of performing those functions necessary to create and analyze a complete PRA. This program includes functions to allow the user to create event trees and fault trees, to define accident sequences and basic event failure data, to solve system and accident sequence fault trees, to quantify cut sets, and to perform uncertainty analysis on the results. Also included in this program are features to allow the analyst to generate reports and displays that can be used to document the results of an analysis. Because this software is a very detailed technical tool, the user of this program should be familiar with PRA concepts and the methods used to perform these analyses. Although SAPHIRE has been designed to be user friendly and makes the process of performing a PRA easier, the complexity of this type of analysis requires a user with a more detailed understanding of PRA concepts than is required by other tools in this suite. The SAPHIRE reference manual and tutorial are available as NUREG/CR-6116, Volumes 2 and 3, respectively. In addition, a technical document that provides information on the principles and algorithms used in the construction and operation of IRRAS and SARA is available as NUREG/CR-6116, Volume 1.

Also included are graphical tools developed for performing risk assessment. These tools include the graphical fault tree, event tree, and P&ID editors. The fault tree editor allows the user to construct and modify graphical fault trees. The event tree editor allows the analyst to construct and modify graphical event trees. The P&ID editor allows the user to construct and modify plant drawings. These drawings can then be used to document the modeling used in a PRA. These editors are an integral part of a PRA. With the graphical editors, the user need not be concerned with the complexity of the SAPHIRE program if the need is only to generate one of these graphical displays.

Introduction

The Integrated Reliability and Risk Analysis System (IRRAS) software development project was started as a result of a recognized need for microcomputer-based software to aid the probabilistic risk assessment (PRA) analyst. The initial scope of the project was to provide a software package that could demonstrate the feasibility of using the microcomputer as a workstation for performing PRA analyses. This package did not necessarily need to perform all of the functions required; however, it did need to provide certain essential functions such as fault tree construction, failure data input, cut set generation, and cut set quantification.

At about the same time, the need for a simple tool that used the results of a PRA to perform limited review and sensitivity analyses was identified. This tool need not be able to create and solve fault trees and event trees, but should be able to perform limited modifications to failure data and cut sets and compare these changes to a base case set of data. This need resulted in another software development project, the System Analysis and Risk Assessment (SARA) system. The IRRAS and SARA system soon became complementary tools for the performance of PRAs. For each release of the IRRAS system there was a corresponding SARA system. The first version of these software packages was released in February of 1987 and contained only the essential concepts mentioned above.

Version 1.0 of IRRAS/SARA was an immediate success and clearly demonstrated not only the tremendous need but also the feasibility of performing this work on a microcomputer. As a result of this success, Version 2.0 development was begun. This package was designed to be a comprehensive PRA analysis package and included all the functions necessary for a PRA analyst to perform his or her work. The areas that were not treated in version 1.0 were addressed, and a complete, integrated package was developed. Because Version 2.0 was a complete rewrite from version 1.0, a thorough test plan was necessary. The major features of Version 2.0 along with an Alpha test were completed in early March of 1988. Following the Alpha test, approximately 15 sites were selected from among the sites currently using Version 1.0. and were sent a Beta test Version 2.0. In May of 1988, the Beta test was completed and work began on fixing any bugs found. In addition, any desired new features that could reasonably be incorporated into version 2.0 were included. Version 2.0 was released in June 1990 and work began on the development of Version 2.5.

Version 2.5 was an integrated PRA software tool that gave the user an enhanced ability to create and analyze fault trees and event trees using a personal computer (PC). This program provided functions for fault tree and event tree construction and analysis. The fault tree functions ranged from graphical fault tree construction to fault tree cut set generation and quantification. The event tree functions included graphical event tree construction, the linking of fault trees, defining accident sequences, generating accident sequence cut sets, and quantifying them.

Version 4.0 contains many significant enhancements over previous versions. This version provides much more powerful cut set generation algorithms. These algorithms are more than a thousand times faster than previous versions. Problems that took hours to solve can now be solved in seconds using Version 4.0. Other enhancements provided in this version include the ability to use the system fault tree logic to solve accident sequences and the addition of flag sets to automatically prune the sequence logic. Many of the operations in IRRAS and SARA have also been streamlined and simplified to provide an even more powerful tool for the PRA analyst. This version has undergone a rigorous testing program to ensure reliability and useability. Overall, Version 4.0 continues to provide more powerful tools for the PRA analyst.

SAPHIRE automates the model creation, manipulation, modification, and quantification processes. Designed for the IBM-PC and compatibles, SAPHIRE is readily accessible and portable. Taking advantage of new state-of-the-art algorithms, SAPHIRE is quite fast and powerful.

SAPHIRE simplifies the analysis process and automates the construction of input to the analysis software. The analyst can graphically construct and modify fault trees. SAPHIRE gives the users better visualization of the fault tree and simplifies the construction and maintenance. The program supports all of the basic constructs involved in fault tree construction, including NOT gates. Once the fault tree is constructed, the program automatically generates the alphanumeric input for the analysis software. The component reliability information is then easily input into the SAPHIRE database using specially designed menus and screens.

IRRAS 4.0 includes fault tree, event tree and cut set editors to improve the analysis capabilities without requiring complete regeneration and reduction of the fault trees. Basic event or initiating event frequencies are easily changed. Cut sets are easily modified with the cut set editor to add recovery actions, or cut sets may be deleted if desired. These changes can be saved in the database and quantified as desired.

Many new features have been incorporated into Version 5.0. These new features include the capability to perform rule based recovery analysis and end state cut set partitioning. A completely new alphanumeric fault tree editor has been developed. This new editor allows the analyst to easily modify fault tree logic. A new cut set editor has also been developed. This editor is much more powerful than the previous editor and the user interface has been simplified. The MAR-D data interchange processor has been completely rewritten to provide much more flexibility in defining output options. An error message file has also been added to allow the analyst to determine the results of batch processing. A windows compatible version of the graphical fault tree editor has been developed and a 386 protected mode version is also now available. These two new versions allow the user to take advantage of the latest in microprocessor technology. The event tree rule editor has been changed to a powerful free format editor. This editor allows the analyst to much more easily define rules for event tree top substitutions. These rules include the ability to define rules for each transfer event tree in an analysis. Associated with the event tree changes is the ability to handle the large event tree, small fault tree methodology better. All these new features combine to make SAPHIRE a powerful PRA analysis tool.

This report provides the SAPHIRE user with a basic understanding of the mathematical and probabilistic concepts needed to understand the basic principles used in SAPHIRE. In addition, it gives an overview of the algorithms used in the program. This report is not intended to provide all of the details some readers may desire. Therefore, references are provided that contain more detail for the interested reader.

The report contains the following topics:

- An introduction to sets and set operations and to the corresponding logical operations
- A review of fault tree construction principles and the philosophy used in SAPHIRE
- An overview of probability theory
- An overview of the cut set algorithms used in SAPHIRE
- A review the quantification techniques used in SAPHIRE
- A summary of the calculation types used for the basic events
- An overview of importance measures
- Discussion of the uncertainty analysis and an introduction to Monte Carlo sampling and Latin Hypercube sampling
- An overview of seismic events
- A list of applicable references
- An example of the details of an SAPHIRE application to a simple fault tree

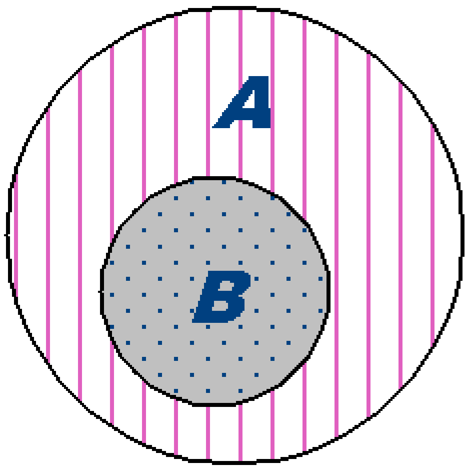
SET THEORETIC AND LOGICAL CONCEPTS

This section presents basic definitions of sets and a summary of useful identities. The reader can obtain more information from Vesely et al. (1981), Mood et al. (1974), or Hahn and Shapiro (1967).

TOPICS

Set Theoretic Concepts

A useful tool to illustrate set relations pictorially is the Venn diagram. Figure 1 shows the Venn diagram for two sets, *A* and *B*, where *B* is a proper subset of *A*.



For SAPHIRE, we are interested in what could occur at a nuclear power plant. Therefore, when set theory is used for SAPHIRE applications, we usually let the population Ω consist of all possible conditions of the plant. Any one element of this set consists of a detailed specification of the condition of every part of the plant. Consequently, Ω has a huge number of elements.

Events are subsets of this population. For example, an event such as "AFW pump PAFWT1 fails to start" is a subset, consisting of all conditions of the pump and its supporting equipment that result in failure to start, together with all possible conditions of the rest of the plant. The event "core melt" is also a subset of the population, containing all the detailed plant conditions that result in core melt.

DEFINITIONS
Set
Population
Null Set
Subset
Proper Subset
Set Equality

Operations on Sets

Three basic operations exist for sets. They are:

union
intersection
complementation

A fourth operation, called set difference , is sometimes considered; it is expressed as a combination of the other set operations.

SEE ALSO
Mutually Exclusive
Exhaustive Sets

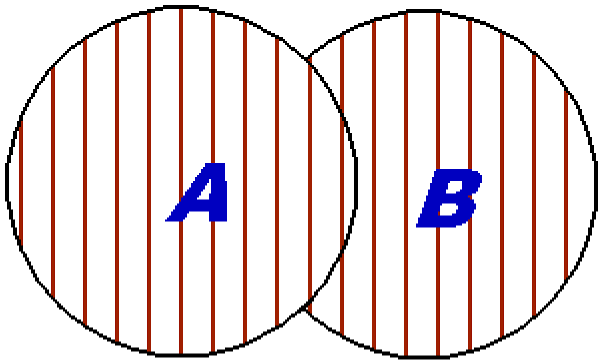
Union

The *union* of two sets is a set consisting of all the distinct elements in *A* or all of the elements in *B* or both. The union operation is also called an OR operation, and is sometimes denoted by $C=A+B$.

The union is denoted by C = A ∪ B

Inexperienced analysts are wise always to use the symbol ∪ to combine sets and the symbol + to combine numbers, but adept symbol jugglers learn to use + safely in both contexts. Computer programs that use only the 128 ASCII characters or the characters on a line printer are forced to use + instead of ∪.

The union of two sets is shown in Figure 2.



The union of any number of sets *A1*, *A2*, ... is the set of all elements that are in any of the *Ai's*. It can be written with notation analogous to summation notation:

$$\bigcup_{i=1}^{n} A_i$$ for n sets and

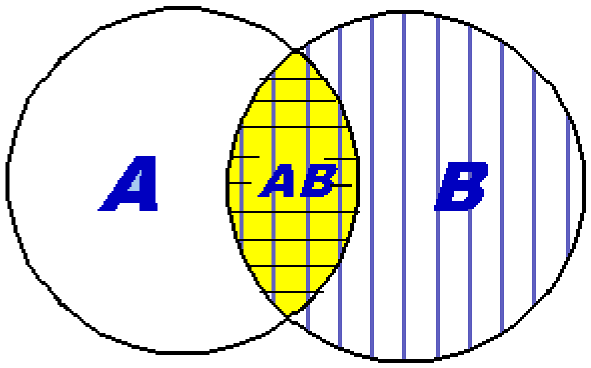
$$\bigcup_{i=1}^{\infty} A_i$$ for infinitely many sets.

Intersection

The *intersection* of two sets is the set consisting of all the elements common to both *A* and *B*. That is, the elements belong to *A* and to *B*. It is also called the AND operation.

The intersection is denoted by C = A ∩ B
or sometimes C = A * B
or simply, C = AB.

The intersection of two sets is shown as the crosshatched region in Figure 3.



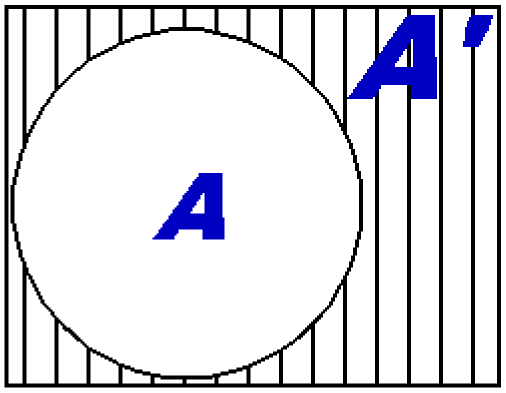
The intersection of *A1, A2,* ... is the set of all elements that are in all the *Ai's*. The intersection of *n* sets can be written as:

$$\bigcap_{i=1}^{n} A_i$$

or, using product notation, as *A1A2...An*.

Complement

The *complement* of a set *A* is the set consisting of all elements in the population that are not contained in *A*. It is sometimes called the NOT operation. It is denoted by *A*', *A*c, or $\tilde{A}$. The complement of a set is shown in Figure 4.



Set Difference

The set of all elements in *A* and not in the set *B* is called the set *difference*.

The set difference is denoted by A ∩ B'

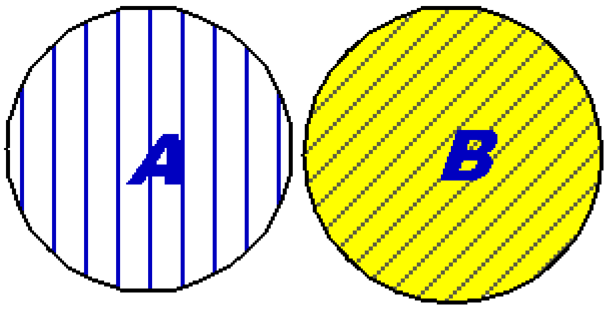
It can also be written as *A-B*. The clear portion of set *A* (shown in Figure 3) represents the set difference *A-B.*

Mutually Exclusive

Two sets are said to be *mutually exclusive* or *disjoint* if and only if they contain no elements in common.

The intersection of mutually exclusive sets is the null set, A ∩ B = ∅.

Mutually exclusive sets are shown in Figure 5.



The sets *A1*, *A2*, ... are mutually exclusive if each pair is mutually exclusive, that is, no element of W is in more than one *Ai*. The term "mutually exclusive" can therefore refer even to an infinite collection of sets.

Exhaustive Sets

A collection of sets *A1*, *A2*, ... is *exhaustive* if the union of the sets is the population Ω, that is, every element of Ω is in at least one *Ai*. In most applications, exhaustive sets are also chosen to be mutually exclusive . When the sets *A1, A2*, ... are both mutually exclusive and exhaustive, they form a *partition* of Ω: every element of Ω is in one and only one of the *Ai's*.

Useful Identities

The following are useful identities in working with sets:

Commutative Laws

$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$

Associative Laws

$$A \cup (B \cup C) = (A \cup B) \cup C$$
$$A \cap (B \cap C) = (A \cap B) \cap C$$

Distributive Laws

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Idempotent Laws

$$A \cap A = A$$
$$A \cup A = A$$

Laws of Absorption

$$A \cap (A \cup B) = A$$
$$A \cup (A \cap B) = A$$

Complementation

$$A \cap A' = A \cap Ac = A \cap \tilde{A} = \varnothing$$
$$A \cup A' = A \cup Ac = A \cup \tilde{A} = \Omega$$
$$(A')' = (Ac)\ c = A$$

Operations Involving Null Set and Population

$$\varnothing \cap A = \varnothing$$
$$\varnothing \cup A = \Omega$$
$$\Omega \cap A = A$$
$$\Omega \cup A = \Omega$$
$$\varnothing' = \varnothing\ c = \Omega$$
$$\Omega' = \Omega\ c = \varnothing$$

DeMorgan's Laws

$$(A \cap B)' = A' \cup B'$$
$$(A \cup B)' = A' \cap B'$$

Other Identities

$$A \cup (A' \cap B) = A \cup B$$
$$A' \cap (A \cup B') = A' \cap B' = (A \cup B)'$$

Statement Logic

A *statement* is defined here as a sentence that can be declared either true or false. Examples are "Generator DG1 fails to start" and "Safety injection is initiated." English statements that are not clearly true or false, such as "This is a nice looking control room," are not considered. Mathematically, a statement is an object that can take one of two values, either TRUE or FALSE. Use the letters *p*, *q*, *r*, etc. to denote statements.

New statements can be built by combining simpler statements using AND, OR, and NOT, defined as follows:

> (*p* AND *q*) is TRUE if both *p* and *q* are TRUE, and it is FALSE if *p* is FALSE, *q* is FALSE, or both are FALSE.
>
> (*p* OR *q*) is TRUE if *p* is TRUE, *q* is TRUE, or both are TRUE. It is FALSE if both *p* and *q* are FALSE.
>
> (NOT *p*) is TRUE if *p* is FALSE, and FALSE if *p* is TRUE.

The symbols of mathematical logic ($\vee$ for AND, $\wedge$ for OR, $^{\rightarrow}$ for NOT) will not be used here. However, for ease of input from a computer terminal, SAPHIRE uses the notation / for NOT. That is /X is the notation for NOT X in SAPHIRE input.

Working from the previous basic definitions, one can prove many simple facts about statements, similar to those listed for sets . For example, one distributive law says

> *p* AND (*q* OR *r*) = (*p* AND *q*) OR (*p* AND *r*)

and one of DeMorgan's laws says

> NOT (*p* AND *q*) = (NOT *p*) OR (NOT *q*).

These equations mean that the statement on the left-hand side is TRUE if and only if the statement on the right-hand side is TRUE. There are many such equations not listed here.

Mathematics that uses the formal manipulation of these logical relations is sometimes called *Boolean*, after the mathematician George Boole.

Set Theory and Statement Logic

Parallel structures for sets and for statements exist: the terms AND, OR, and NOT were used for both, and similar rules such as the distributive laws and DeMorgan's laws applied to both. The relation is made explicit here.

Let Ω be the population, and consider statements about the elements of Ω. Any statement has a corresponding *truth set*, defined as the set of all elements for which the statement is true. An element is in the truth set if and only if the statement is true for that element. For example, the statement "core melt occurs" corresponds to the set of all possible plant conditions that result in core melt. Suppose that

> *A* is the set of elements for which *p* is TRUE
> *B* is the set of elements for which *q* is TRUE.

Then the rules for combining sets and for combining statements are related as follows:

> $A \cup B$ is the set of elements for which (*p* OR *q*) is TRUE
> $A \cap B$ is the set of elements for which (*p* AND *q*) is TRUE
> *A'* is the set of elements for which (NOT *p*) is TRUE.

Because the correspondence is so direct, we sometimes interchange the languages: ***A* OR *B*** instead of $\boldsymbol{A \cup B}$.

For SAPHIRE applications, the statements of interest describe events. For example, the event "AFW pump PAFWT1 fails to start" may be thought of as a statement *p* that can be combined with other statements as described in the *Concepts of Statement Logic* section. The event *occurs* if the statement defining the event is TRUE. This defines an event as a statement. Alternatively, the event can be thought of as naming the set *A* of all plant conditions that result in failure of the pump to start. Similarly, the statement "MOV134 fails to open" can be thought of as corresponding to a set *B* of plant conditions.

> The statement that both these events occur, "MOV134 fails to open AND AFT pump PAFWT1 fails to start," corresponds to the intersection $B \cap A$.

The relation between statements and sets is so direct that most people switch back and forth between the two without even realizing it. This is why the terms AND, OR, and NOT were introduced in *Operations on Sets* as alternative terms for intersection, union, and complementation. The terminology in this document allows for this back-and-forth thinking, not carefully distinguishing between statement logic and set theory.

One reason we did not list all the facts about statements is that they are simply re-expressions of the facts in the *Summary of Useful Identities* section. Any fact about sets in the *Summary of Useful Identities* section can be translated to a fact about statement logic by replacing set symbols with statements:

A, *B*, and *C*	*p*, *q*, and *r*
$\cup$, $\cap$, and $'$	OR, AND, and NOT
the population Ω	always true
the null set ∅	always false

FAULT TREE CONCEPTS

This section provides the reader with an overview of the concepts used by SAPHIRE in the creation of fault tree models. More information can be found in Vesely et al. (1981).

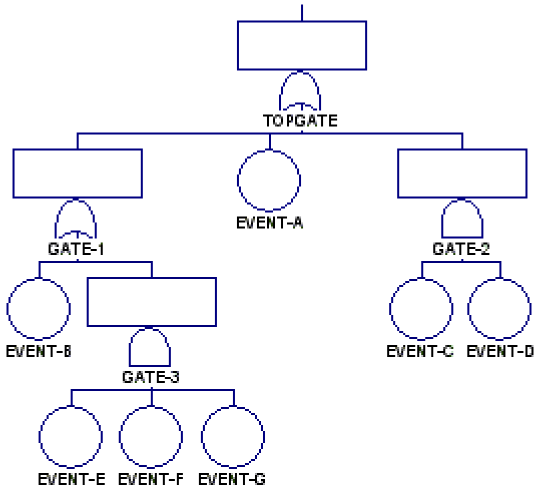
TOPICS

Fault Tree Approach

Fault Tree Symbols

SAPHIRE Fault Tree Approach

SAPHIRE allows the user to input fault tree models in either of two ways: graphically



or alphanumerically

TOPGAT	OR	EVENT-A	GATE-2	GATE-1

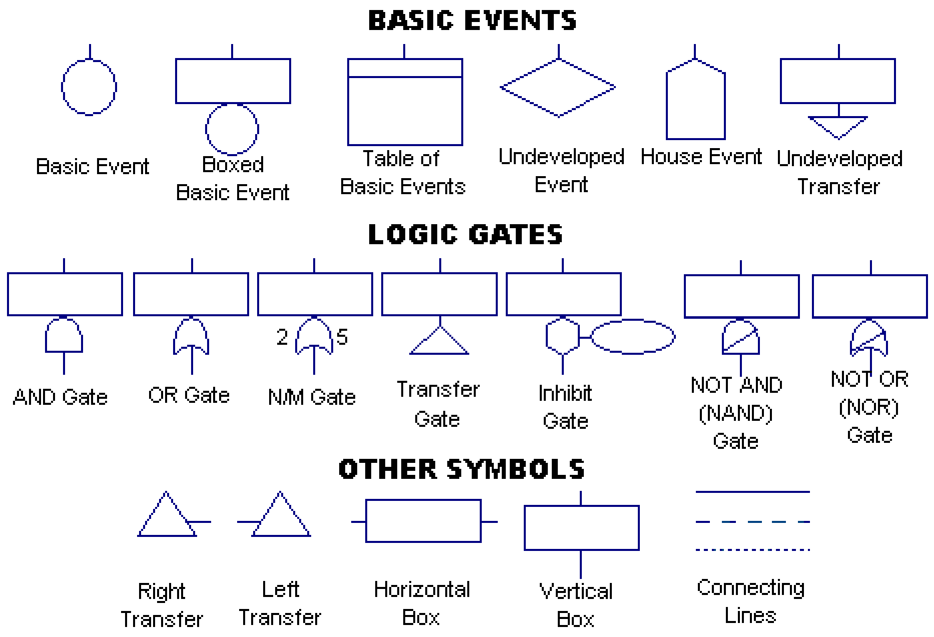
E				
GATE-1	OR	GATE-3	EVENT-B	
GATE-2	AND	EVENT-D	EVENT-C	
Gate-3	AND	EVENT-E	EVENT-G	EVENT-F

Both methods produce equivalent results and use the same basic approach to modeling.

A *fault tree* model consists of a *top event* (usually defined by a heading in an event tree) and a connecting logic structure that models the combinations of events that must take place to result in the undesired top event. **A fault tree is a failure model.** Thus, all the elements in the fault tree generally represent failures, whether they be equipment failures, human errors, or adverse conditions that can contribute to failure of the modeled event. Successful events (those things that should happen) that can contribute to failure of the top event can be included in the fault tree also, but special care must be exercised.

The logic structure must contain only one top event. SAPHIRE will provide an error message if more than one top event is discovered. A simple way to guarantee only one top event per fault tree is to develop the fault tree model from the top down and complete each level of the fault tree model before proceeding to the next level.

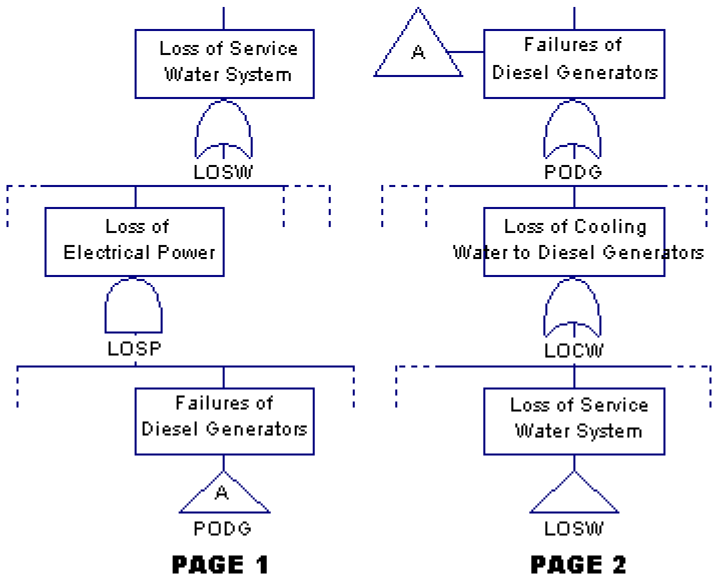
The fault tree logic structure can consist of any combination of the logic symbols shown below that do not result in a logical loop .



Logical Loop

A *logical loop* is a chain of events that comes back on itself.

For example, a service water system can fail due to a loss of electrical power. Part of the electric power model contains failure of the emergency diesel generators. The emergency diesel generators can fail due to a loss of cooling water supplied by the service water system. The combination of events resulting in the loss of service water due to a loss of electrical power caused by failure of the diesel generators that was due to the loss of service water is a logical loop.



Loss of Service Water System
LOSW
Loss of Electrical Power
LOSP
Failures of Diesel Generators
A
PODG
PAGE 1
A
Failures of Diesel Generators
PODG
Loss of Cooling Water to Diesel Generators
LOCW
Loss of Service Water System
LOSW
PAGE 2

This type of circular logic is ambiguous and is not allowed by SAPHIRE. If such a logic pattern is detected, SAPHIRE will provide an error message and will display the sequence of logic gates that are in the loop.

SAPHIRE Fault Tree Symbols

The fault tree model consists of simple faults called basic events and logical operators that dictate how the basic events must combine to result in failure of the fault tree top event. Basic events are the building blocks of the model. When the model is processed, the results will be all the minimal combinations of basic events sufficient to cause failure of the top event. These combinations are called minimal cut sets. Minimal cut sets contain only basic events.

The various fault tree symbols used in SAPHIRE have been grouped into basic events, logic gates , and other symbols. There are six different basic event symbols to indicate different conditions, but all basic events are treated the same in SAPHIRE. The different basic events are:

Basic Event

This represents a simple failure or fault. It may be a hardware failure, a human error, or an adverse condition. Hardware failures are usually expressed in terms of a specific component and a failure mode, such as "Service Water Pump 1A fails to start on demand." Human errors can be failure to carry out a desired task (failure to open a valve), failure to perform a specific recovery action (failure to start a backup system), or execution of a wrong action that has adverse effects on the fault tree top event (isolated the source of water for a cooling system). An adverse condition is not necessarily a failure but in combination with other events can lead to failure. For example, the temperature being below 32°F is an adverse condition necessary for the failure of flow reduction due to a frozen pipe.

Even though a basic event does not necessarily describe a failure, the vast majority of basic events are failures. This leads to loose but understandable language such as "the event is in the failed state" instead of the more correct "the event occurs."

Basic events are always assumed to be independent of each other, in the statistical sense defined in the section on Independent Events . This means that the occurrence of one basic event does not influence the probability of occurrence of any other basic event. For example, suppose that there are two diesel generators, and the failure of either to start on demand is a basic event. Independence of the basic events says that if one diesel generator fails to start on demand, this does not alter the probability that the second diesel generator will fail to start.

A common cause event, such as "two diesel generators fail to start because of unusually cold weather," must be modeled as its own basic event, and be assigned its own failure probability or failure rate. This event is then regarded as statistically independent of all other basic events.

Boxed Basic Event

This event is the same as a basic event except the box provides room to add descriptive text to the event. This does not influence the logic of the fault tree, but adds clarity to the model for those using and reviewing it.

Table of Basic Events

This symbol is a convenience for the modeler. If there are many basic event inputs to a particular logic gate, the events can be listed in a table rather than trying to connect many basic event symbols to the logic gate. This can be done for any logic gate that can receive more than one input. SAPHIRE processes the list of basic events as if they were shown separately. The tradeoff is the inability to add descriptive text to each basic event in the table.

Undeveloped Event

This symbol is used to denote a basic event that is actually a more complex event that has not been further developed by fault tree logic either because the event is of insufficient consequence or because information relevant to the event is unavailable. This event is used by SAPHIRE just like any other basic event.

House Event

A house event is used to denote a failure that is guaranteed to always occur for the given modeling conditions or is guaranteed to never occur for the given modeling conditions. This has unique implications in the processing of the logic model. (See Determination of Minimal Cut Sets for a discussion of how house events impact the logic of the fault tree.)

In the SAPHIRE graphic displays, the house symbol is used mainly for clarity of the model. The determination of whether an event is a house event or not is established when the calculation type is assigned to the basic event . Thus, any basic event in SAPHIRE can be made into a house event.

Undeveloped Transfer

This symbol indicates that the event is complex enough to have its own fault tree logic developed elsewhere; however, to simplify the present fault tree, the event will be treated as a basic event . Usually the complex event is processed as a separate event tree and the results are used as the failure probability for the representative basic event. This can greatly simplify a large fault tree, speeding up processing time. However, with the current capabilities of SAPHIRE, there is little advantage to this technique. It is presented in SAPHIRE because many existing models being transferred from other software into SAPHIRE use it.

Logic Gates

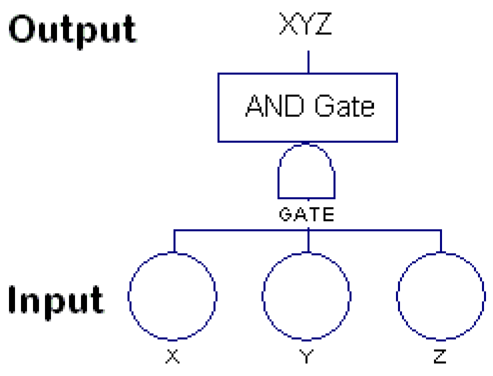
Logic gates are used to indicate how the basic events must combine to result in failure of the top event. Every logic gate has one or more inputs at the bottom and an output at the top. Inputs may be basic events or other logic gates. The output must serve as the input to another logic gate or result in the top event. Each logic gate derives its name from the manner in which the inputs must combine to pass through it to the next level. The input to a logic gate is a set of events. The output is a single event, formed by using the set operations AND and OR on the input events.

The logic gates in SAPHIRE are:
AND Gate
OR Gate
N/M Gate
Transfer Gate
Inhibit Gate
NOT AND Gate
NOT OR Gate

Other symbols:
Right (Left) Transfer
Horizontal (Vertical) Box
Connecting Lines

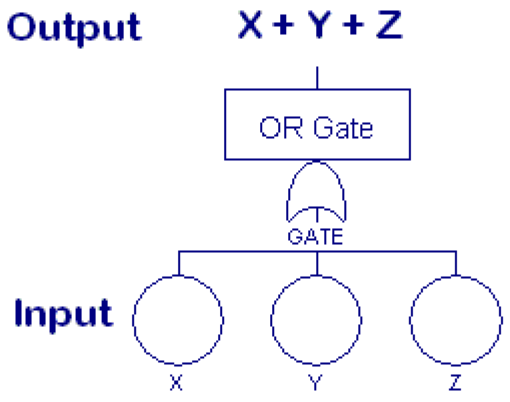
AND Gate

This gate states that the output event is the simultaneous occurrence of all the input events. In set language, the output set is the intersection of the input sets. In terms of statement logic, the output is a compound statement (*X* AND *Y* AND *Z*).



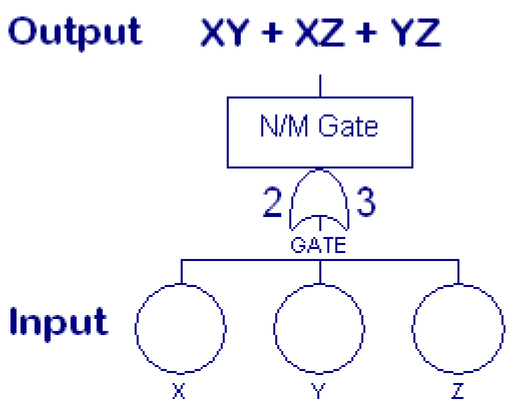
OR Gate

This gate combines the inputs by the OR operation. The output set is the union of the three input sets. Alternatively, the output statement is *X* OR *Y* OR *Z*.



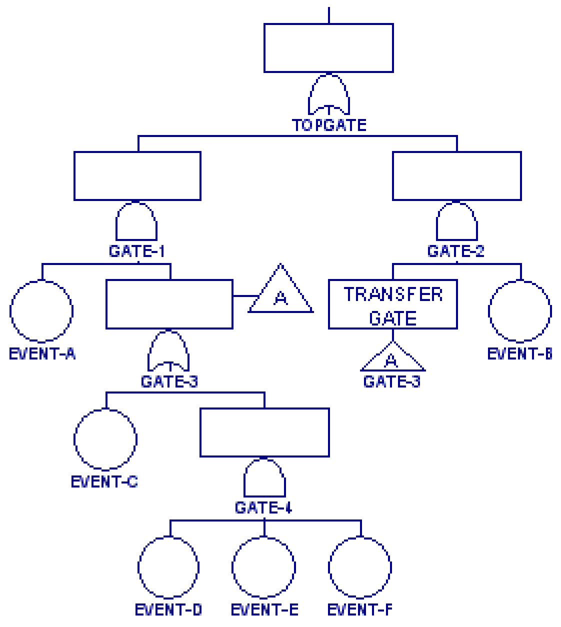
N/M Gate

This gate states that N of the M input events occur. It is sometimes called an N-out-of-M gate or a *combination gate*. For a 2/3 gate, illustrated here, 2 of the 3 input events must occur. The output statement is (*X* AND *Y*) OR (*X* AND *Z*) OR (*Y* AND *Z*).



Transfer Gate

This gate does not require any special logic to result in an output, rather it is used to link logic structures together without introducing any new logic of its own. This is used primarily as a convenience for the modeler. All but the simplest of fault trees take up more than one page. The TRANSFER GATE indicates where the logic on a given page is continued on another page. A TRANSFER GATE may also be used to indicate where the logic is continued on the same page. This is shown in Figure 13, where GATE-3 is an input both to GATE-1 and to GATE-2. In SAPHIRE, the following rules apply when using a TRANSFER GATE:



- The TRANSFER GATE name must be the same as the name of the gate where the logic continues.

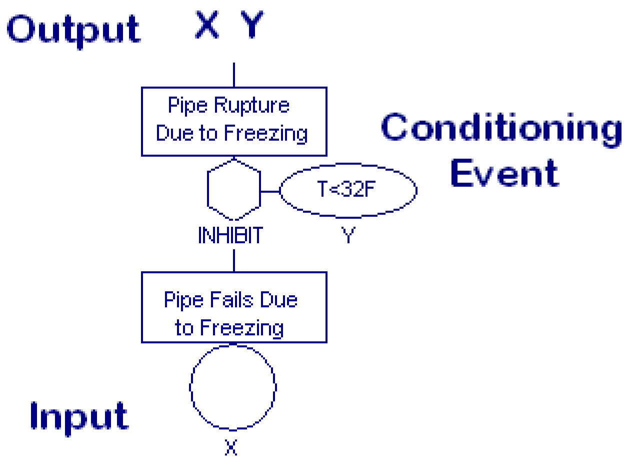
- When transferring on the same page, the gate being transferred to can be anywhere on the page, except where it would create a logic loop .

- When transferring to another page, the gate being transferred to must be the top gate on the page.

- When transferring to another page, the transfer gate name, the file name for the page being transferred to and the name of the gate being transferred to must all be the same. For example, if the TRANSFER GATE is called TRANS1, then the page being transferred to must be called TRANS1 and the top gate on that page must be called TRANS1.

Inhibit Gate

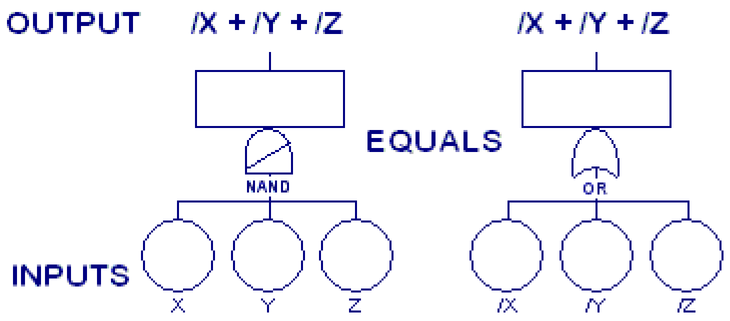
This gate, as its name implies, has its output inhibited unless a certain condition is met. The output event occurs if the single input fault occurs in the presence of an enabling condition. The input event is connected to the bottom of the gate and the conditioning event is drawn to the right of the gate.



Event X cannot occur unless Conditioning Event Y is present. The output is the combination of events X and Y. Thus, the INHIBIT GATE is a special type of AND GATE and SAPHIRE processes it as such. The Conditioning Event is treated as any other basic event with a probability of occurrence calculated and used in the processing.

NOT AND Gate

This gate is also called a NAND GATE. It can be thought of as the negation of an AND GATE. The output occurs if any one of the inputs does not occur. This is best explained through an example.



The left side of Figure 15 shows a NOT AND GATE with inputs X, Y, and Z. If any one of the inputs does not occur, then an output occurs. Any of three possibilities satisfy this condition: 1) X does not occur, 2) Y does not occur, or 3) Z does not occur. Since any event (X) and its complement (/X) are mutually exclusive, we can say that

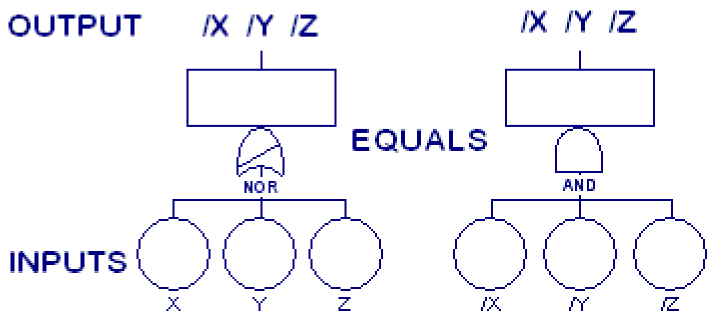
X does not occur = /*X* occurs.

Therefore, the output of the NOT AND GATE in Figure 15 is /*X* (read not *X*), or /*Y*, or /*Z*.

Another way of looking at the problem is the way SAPHIRE actually processes a NOT AND GATE. The gate is transformed into an OR GATE with all of the inputs transformed into their complements. This is shown on the right side of Figure 15. Any single complement event occurring results in an output.

NOT OR Gate

This gate is also called a NOR GATE. It is the negation of an OR GATE. The output occurs if none of the inputs occur. This is shown in Figure 16. There is only one combination of events where none of the inputs occur; *X* does not occur and *Y* does not occur and *Z* does not occur. In terms of complemented events this is /*X* and /*Y* and /*Z*.



SAPHIRE processes a NOT OR GATE by transforming it into an AND GATE with all of the inputs transformed into their complements. All of the not events must occur for the output event to occur. This is the same as none of the original events occurring. The other symbols in a fault tree are used to add clarity to the diagram and to connect the various gates and events together properly.

Right or Left Transfer

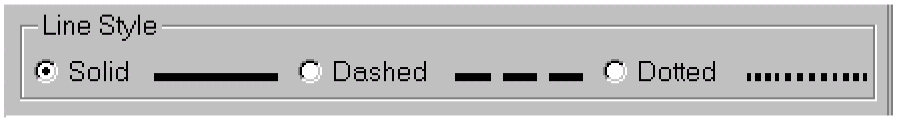
These symbols are used to indicate where a transfer has taken place. At the place where the original line of logic left off is a TRANSFER GATE. At the place where the logic picks up again, a RIGHT or LEFT TRANSFER symbol is placed. This makes it easier for a reader or reviewer to follow the logic through a large fault tree taking up several pages. Typically, the TRANSFER GATE and its corresponding TRANSFER symbol are given the same label, as shown in Figure 13 .

The RIGHT (LEFT) TRANSFER symbol is strictly for reader convenience and is not needed by SAPHIRE to have a correct model. SAPHIRE has all the information it needs from the TRANSFER GATE name and fault tree page file name to generate the proper logic. The presence or absence of a transfer symbol is ignored by SAPHIRE.

Horizontal or Vertical Box

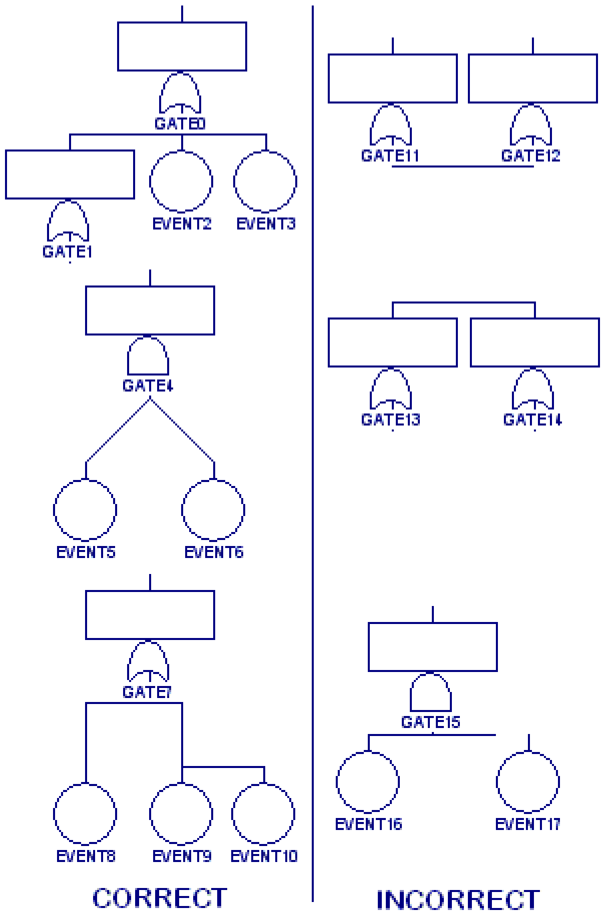
These boxes are also provided for the convenience of the reader/reviewer. They allow further descriptive information to be placed in the diagram than that contained in the boxes attached to the various gates and events. SAPHIRE ignores these boxes when processing the fault tree.

Connecting Lines



Line Style: [✓ Solid] Dashed Dotted

Three line types are provided in SAPHIRE. As shown, these are a solid line, a dashed line, and a dotted/dashed line. The different line types can be used to highlight or differentiate various portions of the fault tree model. All three line types are treated the same by SAPHIRE. Lines are used to connect the gates and basic events together to form the logic of the fault tree. A single input can be attached to a gate directly without using any line. If there is more than one input to a gate, then a line or table of events must be used to make the connection. Lines may be drawn at any angle. Connecting lines must actually touch the symbols being connected and must do so at the input or output stems on the symbols. Events or gates left dangling will not be part of the fault tree logic. Lines always connect outputs to inputs, never input to input or output to output. Figure 17 shows examples of correct and incorrect use of lines.



PROBABILITY CONCEPTS

This section provides the reader with an overview of the concepts of probability associated with the uncertainty analysis used in PRA. This discussion will not be inclusive, but it will present the basic concepts and principles. For a more detailed discussion of these topics, the reader can obtain more information from Press (1989), Lindley (1985) and Singpurwalla (1988).

TOPICS

Definition of Probability

Probability is the only satisfactory way to quantify our uncertainty about an uncertain event E. Probability is always *conditional*; it is conditioned on all of the background information we have at the time we are quantifying our uncertainty. This background information is denoted by H and the probability of E conditional on H is denoted by P(E|H). To make the notation less cumbersome, we write this simply as P(E); nevertheless, the conditioning H should be understood.

The range of a probability is between 0 and 1. P(E) = 0 means E will never occur, and P(E) = 1 means E will always occur. From now on, assume that a probability is defined for all events in the population.

Rules of Probability

The rules of probability tell us how to relate our uncertainty about events. Specifically, they tell us how various probabilities combine or cohere. These rules are motivated by preferences between events and a scoring rule argument. The scoring rule approach can be used to show that the following three rules of probability hold for discrete cases.

For any event,

$$\mathbf{0 \le \mathit{P(E)} \le 1, \mathit{and}\ \mathit{P}(\Omega) = 1} \qquad (4\text{-}1)$$

For any mutually exclusive events E1, E2, ...

$$P\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} P(E_i) \qquad (4\text{-}2)$$

The *conditional probability* of an event F given an event E is

$$\boldsymbol{P(F|E) = P(F \cap E) \,/\, P(E)} \qquad (4\text{-}3)$$

which is equivalent to the multiplication rule

$$\boldsymbol{P(F \cap E) = P(F|E)\, P(E)}.$$

These are the basic rules of probability, from which all others can be derived. One logical development of probability, due to Kolmogorov (Press 1989), takes Equations (4-1) and (4-2) as axioms, and Equation (4-3) as a definition. A more recent approach by Renyi (Press 1989) uses conditional probability as the fundamental concept, rewrites every unconditional probability above as a conditional one, and uses the rewritten Equations (4-1), (4-2) and (4-3) as axioms. These mathematical fine points are not important to this report. It is enough to note that every treatment of probability uses the rules given above, and the rules that follow as consequences in the sections below.

Equation (4-2) says that the probability of the union of disjoint events is the sum of the probabilities. This fact motivated the use of **+** as an alternate notation for ∪ in Section 2.2.1.

Law of Total Probability

For any events E and F,

$$P(E) = P(E \cap F) + P(E \cap F') = P(E|F)\, P(F) + P(E|F')\, P(F')$$

This law can be extended to a set of *n* mutually exclusive and exhaustive events *F1*, *F2*, . . . , *Fn* as follows:

$$P(E) = \sum_{k=1}^{n} P(E|F_k) P(F_k) \qquad (4\text{-}4)$$

Basic Probability Relations

$$P(\Omega) = 1$$

$$P(\emptyset) = 0$$

$$P(\bar{A}) = P(A') = 1 - P(A)$$

$$P(A \cup \bar{A}) = P(A \cup A') = P(\Omega) = 1$$

$$P(A \cap \bar{A}) = P(A \cap A') = P(\emptyset) = 0$$

If E and F are two events and E is a subset of F, then $P(E) \leq P(F)$.

Bayes' Law

Consider any two events E and F. By the multiplication law

$$\boldsymbol{P(E \cap F) = P(E|F)\, P(F) = P(F|E)\, P(E).}$$

so

$$P(E|F) = \frac{P(F|E)P(E)}{P(F)} \qquad (4\text{-}5)$$

We use Equation (4-5) to change our uncertainty about E given background information H to our uncertainty about E given F and H. We can think of F as new data.

For example, suppose that turbine-driven pumps fail to start with some frequency *p*. We quantify our background knowledge about turbine-driven pumps through a probability distribution on *p*. (For ease of explanation, suppose that this distribution is discrete, a list of possible values *p*i, each with a probability reflecting our degree of belief.)

To continue this example, let E be the event "$p = 0.01$". Let F be the event "3 failures in 100 attempts to start." We know P(E) from the probability distribution that quantifies our background knowledge. How should this probability be changed to account for the new information? That is, what is P(E|F)?

This question is answered using Bayes' Law. The theory of binomial random variables shows that

$$P(F;p) = \binom{100}{3} p^3 (1-p)^{100-3}$$

is the probability of the event F given some value of *p*. Therefore P(F|E) is P(F;*p*) with the value 0.01 substituted for *p*. The value of P(F) is obtained from the law of total probability, Equation (4-4):

$$P(F) = \Sigma[P(F;p_i)P(p = p_i)]$$

summed over all the possible values *p*i. Then finally, P(E|F) is obtained by substituting the values for P(E), P(F|E), and P(F) into Equation (4-5).

In summary, we used Equation (4-5) to change a belief about E given the background information to a belief about E given both the background information and F. The belief was updated based on new data.

Independent Events

We say an event E is *independent* of another event F if the probability of E, P(E), is unaltered by any information concerning event F. We write

$$\boldsymbol{P(E|F) = P(E|F') = P(E)} \ .$$

This is also called *statistical independence*. From this definition we obtain the following relationship for independent events

$$\boldsymbol{P(E \cap F) = P(E|F)P(F) = P(E)P(F)} \ .$$

Beginners often confuse mutually exclusive events with independent events. The two concepts are different: Mutually exclusive events satisfy

$$\boldsymbol{P(A \cap B) = P(\varnothing) = 0}$$

whereas independent events satisfy

$$\boldsymbol{P(A \cap B) = P(A)\ P(B)} \ .$$

Therefore, if *P*(*A*) > 0 and *P*(*B*) > 0, *A* and *B* cannot be both mutually exclusive and independent. The mutual exclusiveness introduces a dependence: if *A* occurs, *B* cannot occur.

Additional Probability Relations

The probability of the union of n events is

$$P(A_1 \cup A_2 \cup \cdots \cup A_n) = \Sigma P(A_i) - \sum_{i<j} P(A_i A_j) + \cdots + (-1)^n P(A_1 A_2 \cdots A_n) \qquad (4\text{-}6)$$

The probability of the intersection of n events is

$$P(A_1 A_2 \cdots A_n) = P(A_n | A_1 \cdots A_{n-1}) \cdots P(A_2 | A_1) P(A_1) \qquad (4\text{-}7)$$

The probability of the intersection of n events when the events are statistically independent is

$$P(A_1 A_2 \cdots A_n) = P(A_1)P(A_2) \cdots P(A_n) \qquad (4\text{-}8)$$

For any n events (dependent or independent), we have

$$P(A_1 A_2 \cdots A_n) \leq \min[P(A_1), P(A_2), \ldots, P(A_n)] \qquad (4\text{-}9)$$

For independent events, the probability of the intersection equals the product of the probabilities. This fact motivated the product notation that was introduced as an alternate to $\cap$ in the discussion on the intersection of sets. Because of its compactness, the product notation has been used for intersections in Equations (4-6) through (4-9).

DETERMINATION OF MINIMAL CUT SETS

When considering the development of a fault tree minimal cut set algorithm, it is good to review the general processes involved. First, we have the definition of the fault tree logic. Typically, the logic is defined using an alphanumeric file containing names of gates and basic events. Gate and event names vary in length, but 16 characters seem to be a typical size. Along with the logic file is another alphanumeric file containing basic event names and a failure probability associated with each event. These failure probabilities are used during the fault tree solution process to simplify the tree by truncation. Additional processing information may be used, but this is typically the minimum information required.

The above information is loaded into memory and converted into a format that is easier to process. Names are usually converted to numbers for smaller size and ease of manipulation. Certain optimization functions are also performed on the logic before it is processed. Next, the logic for each gate starting with the TOP is recursively replaced with its inputs until the resulting logic is in terms of basic events only. This results in a list of event intersections. Each event intersection is a *cut set* of the fault tree and identifies a set of events that will cause the function modeled by the fault tree to occur. The list of cut sets identifies all the logical combinations of events that will cause the top event to occur.

The cut sets described above may need further reduction due to rules defined for Boolean reduction. These reductions are applied to obtain a simpler collection of cut sets. For example, the cut sets generated should be *minimal*, that is, the list should not be simplifiable.

For example, if $A \cap B \cap C$ causes the top event to occur, then $A \cap B \cap C$ is a cut set. If $A \cap B$ is also a cut set, then $A \cap B \cap C$ is not minimal, and it is discarded from the list. If neither A alone nor B alone causes the top event to occur, $A \cap B$ is a minimal cut set, and it is retained in the list. This is an application of the absorption identity:

$$(A \cap B) \cup (A \cap B \cap C) = A \cap B.$$

The event probabilities are then used to calculate a probability for each cut set using Equation (4-7). This value is the probability that the given set of events will occur. Any cut set whose probability falls below a user-defined value is then eliminated. The remaining cut sets are the minimal cut sets for the fault tree and are the desired end product of the fault tree solution. In SAPHIRE, the minimal cut sets are always in terms of basic events unless the analyst specifically indicates that certain gates are to be treated as basic events.

Once the minimal cut sets have been determined, the quantification routines must be employed to determined a point estimate for the probabilities of the cut sets. The routines that find importance measures would then be used to calculate the importance of each basic event in the cut sets, and the uncertainty routines would be used to perform uncertainty analysis on the cut sets.

The steps described above need not be applied in the order indicated, but each step is usually present in any fault tree software. We will now present a more detailed overview of each of these steps as they relate to SAPHIRE.

In order to solve a fault tree, there are a number of operations that must be performed on the tree before it can be solved. Some of these operations relate to converting the tree into a format that is ready to solve, while others involve optimizing the tree to make the processing of the tree more efficient.

SEE ALSO

Recursive Algorithms
Loading and Restructuring
N/M Gate Expansion
TOP Gate Determination
Loop Error Detection
Complemented Gate Conversion
House Event Pruning
Coalescing Like Gates
Modules Versus Independent Subtrees
Module Determination and Creation
Independent Event Determination
Independent Gate and Subtree Determination
Determining Gate Levels
Fault Tree Reduction
Cut Set Truncation
Intermediate Result Caching
Fault Tree Cache Initialization
Fault Tree Gate Expansion
Cut Set Absorption

Boolean Absorption
Data Storage Considerations
Sequence Cut Set Generation

Recursive Algorithms

Many of the processes associated with fault tree reduction and quantification can be implemented easily using recursive procedures. A simple definition of a *recursive* procedure is "a procedure that calls itself."

An example of where a recursive procedure might be used is in checking a gate for "valid" inputs. A recursive implementation of this procedure has as an argument, the gate to be checked. This procedure checks each input to the gate passed as an argument. If an input is a basic event, then it checks to see if it is valid. If the input is a gate, however, it calls itself to see if the inputs to this gate are valid. When all the inputs to a gate have been processed, the procedure exits and continues processing the gate it was checking before the recursive call. The algorithm stops when all inputs to all gates have been checked.

Many computer languages do not support recursive procedures, but in those languages recursion can be simulated by using arrays to keep track of the arguments passed to the procedure. SAPHIRE takes advantage of recursive procedures in many areas.

Loading and Restructuring

SAPHIRE was designed to allow the user to structure very large fault trees into smaller pieces or pages. The concept of pages comes from the graphical fault tree editor. One page represented the portion of a fault tree that could be easily displayed on a graphical screen or printed on a standard sheet of paper. This idea expanded to allow the pages of the fault tree to be connected together with transfer gates. SAPHIRE stores fault trees by pages, in a relational database. The name of each system is the key to locate the system (fault tree) in the database. Transfer gates are stored as subsystems. Again, the name of the transfer gate is the name of the subsystem. During the load process, these names are used to connect the fault tree logic.

Because SAPHIRE stores the logic of these fault trees as physically separate pages, connected by transfer gates, the first task is to load these pages into memory and combine them into one connected fault tree. This is done by reading in the logic for the first page of the tree, then recursively scanning the loaded logic for a transfer gate that has not been processed. SAPHIRE allows the user to specify whether a transfer gate is to be expanded or not. The gates that are flagged (identified as not to be expanded) are converted to basic events at this time.

During the load process, SAPHIRE connects gates to the tree by name. The gates are maintained in a sorted list that is searched using a binary search, when required. When a new gate is encountered, it is inserted into the gate list in sorted order. As the tree is loaded, transfer gates are replaced by gates with developed logic beneath them. During this process, if

SAPHIRE encounters a gate that is not a transfer and has the same name as another gate, it checks to see if it is an identical gate (i.e., it is the same type and has the same inputs). If the gates are not identical, SAPHIRE displays an error message and terminates the process after the tree is loaded.

When all transfer gates have been processed, any transfer gates remaining are considered to be unresolved transfer gates. The user is notified of these and they are converted to basic events with the same name as the transfer gate. This allows SAPHIRE to continue processing the fault tree. These unresolved transfers will appear as basic events in the cut sets.

If the tree is successfully loaded, SAPHIRE checks to see if the user has specified a gate name to be used as the top gate. If so, then the tree is pruned to eliminate any logic that is not connected beneath this gate. This process simplifies the tree and frees any memory used by the excess logic. At this point, the tree is ready for further processing.

N/M Gate Expansion

The next step is to convert N/M gates to their representative logic in terms of AND and OR gates. This type of gate is used in SAPHIRE to simplify the definition of the logic for situations where the user needs to define a structure representing the combination of *M* things taken *N* at a time. The user may specify any combination where *N* and *M* range from 2 to 9 and *N*<*M*. SAPHIRE automatically converts these gate structures by first generating a number of intermediate AND gates containing as inputs the combinations of inputs represented, then these gates are input to the original N/M gate. Once this is complete, the N/M gate type is changed to an OR gate. The number of AND gates under the OR gates is determined by the total number of combinations of N failures out of a population of *M* events. The equation for this number of combinations is

$$\binom{M}{N} = \frac{M!}{N!(M-N)!}$$

An example of this process can be illustrated with the following "2/3" gate.

GATE1	2/3	INPUT1	INPUT2	INPUT3

is converted to the following structure:

GATE1	OR	N/M-1	N/M-2	N/M-3
N/M-1	AND	INPUT1	INPUT2	
N/M-2	AND	INPUT1	INPUT3	
N/M-3	AND	INPUT2	INPUT3	

Thus, for 2 out of 3 gates, there are 3 unique combinations of 2 failures. This generates 3 AND gates under the OR gate. If the number of inputs to the gate does not equal *M*, then a fatal error message is generated. In this case, SAPHIRE will not try to solve the fault tree.

TOP Gate Determination

If the user has not specified the gate to be used as the top gate of the fault tree, the next step in solving the fault tree is to determine which gate is the "TOP" gate. This is done by counting the references to each gate. A gate is referenced if it appears as input to any other gate. The top gate is the only gate that will not be referenced by any other gate. If SAPHIRE detects more than one gate that qualifies as the TOP gate, then the user is notified and given the opportunity to select the gate to be used as the TOP gate. If no gate is selected, SAPHIRE will not try to solve the fault tree. If, however, the user selects one of the gates, SAPHIRE will prune all other logic not connected to this gate and continue with the solution.

Loop Error Detection

Now that the TOP gate of the fault tree has been determined, SAPHIRE can proceed to check for loops in the fault tree. A loop is a situation where a gate either directly or indirectly references itself. A simple example of a loop is represented by the following fault tree logic:

TOP	AND	GATE1	EVENT1	
GATE1	OR	GATE2	GATE3	EVENT2
GATE2	OR	EVENT3	EVENT4	
GATE3	AND	GATE1	EVENT5	

In this example, GATE1 indirectly references itself since GATE1 references GATE3, and GATE3 references GATE1.

To determine if there is a loop in the fault tree logic, SAPHIRE defines a Boolean array containing one element for each gate in the fault tree. This list is then initialized to FALSE. During processing of a gate, the Boolean variable for that gate is TRUE when processing that gate or any of its inputs, otherwise it is FALSE. Starting with the TOP gate, SAPHIRE traverses the fault tree by following the gates defined in the inputs to each gate. As a gate is encountered, its Boolean variable is tested. If the value of this variable is TRUE, then a previous reference to this gate must have occurred indicating a loop exists in the fault tree at this point. If Boolean variable is FALSE, then it is set to TRUE to indicate that this gate is currently being processed and the inputs for this gate are traversed. When all the inputs to a gate have been checked, the Boolean variable for the gate is set to FALSE before exiting. Using the previous loop example, the processing proceeds as follows:

(1) Initialize Boolean array.

TOP	**GATE1**	**GATE2**	**GATE3**
FALSE	FALSE	FALSE	FALSE

(2) Start processing the TOP gate.
Set flag for TOP gate.

TOP	**GATE1**	**GATE2**	**GATE3**
TRUE	FALSE	FALSE	FALSE

(3) Process the first input to the TOP gate.
First input is GATE1.
Set flag for GATE1 and continue.

TOP	**GATE1**	**GATE2**	**GATE3**
TRUE	TRUE	FALSE	FALSE

(4) Process the first input to GATE1.
First input is GATE2.
Set flag for GATE2 and continue.

TOP	**GATE1**	**GATE2**	**GATE3**
TRUE	TRUE	TRUE	FALSE

(5) No gates input to GATE2.
Reset flag for GATE2 and exit.

TOP	**GATE 1**	**GATE2**	**GATE3**
TRUE	TRUE	FALSE	FALSE

(6) Continue processing inputs to GATE1.
Next input is GATE3.
Set flag for GATE3 and continue.

TOP	**GATE 1**	**GATE2**	**GATE3**
TRUE	TRUE	FALSE	TRUE

(7) Process inputs to GATE3.
First input is GATE1.
Set flag for GATE1.
Flag is already set.
Loop detected!

TOP	**GATE 1**	**GATE2**	**GATE3**
TRUE	TRUE	FALSE	TRUE

Two points of optimization can be considered in this approach. First, each gate only needs to be processed once. If it is referenced several times in the fault tree, repeated processing can be time consuming. SAPHIRE maintains a list of those gates that have been processed and only traverses those that have not been previously processed. Second, this algorithm is quite repetitive and can be implemented quite nicely as a recursive procedure (see Section 5.1).

If SAPHIRE detects a loop in the fault tree, a fatal error is generated along with a traceback. This traceback defines exactly the gate reference list that caused the loop. SAPHIRE will not process a fault tree that has loops. The user must modify the logic to remove the loop before SAPHIRE will solve the fault tree.

Complemented Gate Conversion

Once SAPHIRE has ensured that the fault tree logic does not contain any loops, the complemented gates in the fault tree are processed. Two types of complemented gates are allowed in SAPHIRE. The user may indicate a complemented gate by using either the NAND or the NOR gate or by putting a forward slash (/) in front of a gate name. If the complemented gate types are used, then all references to the gate name will use the complemented logic. If the user wants to complement only a specific reference to a gate, then the slash character may be used in front of the gate name where it is referenced.

SAPHIRE processes complemented gates by first complementing the gate type, then complementing the inputs to the gate. The following example demonstrates this process:

TOP	AND	GATE1	GATE2
GATE1	NAND	GATE3	EVENT1
GATE2	AND	GATE3	EVENT2
GATE3	NOR	EVENT3	EVENT4

becomes

TOP	AND	GATE1	GATE2
GATE1	OR	/GATE3	/EVENT1
GATE2	AND	GATE3	EVENT2
GATE3	AND	/EVENT4	/EVENT5

where the "/" character represents the complement of the input.

Notice that GATE3 is referenced as both a complemented gate and a noncomplemented gate. To handle this, SAPHIRE generates a new gate called NOT3 that contains the complemented version of GATE3. Now, the new fault tree is as follows:

TOP	AND	GATE1	GATE2
GATE1	OR	NOT3	/EVENT1
GATE2	AND	GATE3	EVENT2
GATE3	AND	/EVENT4	/EVENT5
NOT3	OR	EVENT4	EVENT5

If every gate in the tree is referenced in the fault tree as both complemented and noncomplemented, then this approach to processing the complemented gates can result in a fault tree with twice the number of gates as in the original tree. This, however, is not usually the case and the number of additional gates is substantially smaller. When SAPHIRE first encounters a reference to a complemented gate in the fault tree, it assumes that this will be the only reference to the gate, therefore, it complements the original gate. If later on it encounters a reference to the noncomplemented version of the gate, it then generates a new gate that is identical to the original uncomplemented gate.

House Event Pruning

SAPHIRE allows the user to modify the logic structure of a fault tree by using "house" events. House events are events that can be set to logical TRUE (T) or FALSE (F). This forces the event to occur with house event TRUE, or forces it not to occur with house event FALSE. SAPHIRE also allows the user to specify that an event is to be ignored with house event IGNORE (I) which says to remove the event from the fault tree logic. An event set to house event IGNORE will be treated as if it did not exist in the fault tree.

Normally, house events are treated as special events that must be designated as house events. In SAPHIRE, however, the user may treat any event as a house event. Since SAPHIRE creates an event for each transfer gate in the tree, house events may also be used to prune subsystems from a fault tree. At various times, SAPHIRE will use house events to simplify or optimize the processing of the fault tree. There are two of these situations. First, if the user is truncating on probability and the probability of an event is below the truncation value, then we know that this event has negligible probability of occurring. To prune the fault tree, we set these events to house event FALSE. This same technique could be used for other truncation criteria that can be determined before the fault tree is solved to further simplify the tree.

Second, SAPHIRE uses house events when solving sequence cut sets. In SAPHIRE, accident sequences are defined using an event tree to indicate the failure or success of top events. Each top event in the event tree is associated with a system fault tree (see Section 5.22). To solve the accident sequence, SAPHIRE constructs a fault tree for those systems that are defined to be failed in the sequence logic by creating a dummy AND gate with these systems as inputs. SAPHIRE then solves this fault tree using the specified truncation values. This process results in a list of cut sets for the failed systems in the accident sequence. SAPHIRE then uses the "cut set matching" technique to further reduce this list of failed system cut sets. This technique uses the cut sets determined from solving the successful system fault trees in the accident sequence logic to eliminate cut sets from the list of failed system cut sets. To do this, SAPHIRE first scans the list of failed-system cut sets and assigns a value of FALSE to any event in SAPHIRE that does not appear in this list. Once this is done, the fault tree representing the successful systems in the accident sequence logic is constructed, pruned by the house events, and solved. The events that are set to FALSE in the previous step result in a significantly reduced success system fault tree. We can do this since we know that for any successful-system cut set to eliminate a failed-system cut set, it must contain only events in the list of failed-system cut sets. Setting these events to house event FALSE will ensure that the cut sets with these events in them will be eliminated at the fault tree restructuring step. This process greatly speeds up the solution of the successful system fault tree. For example, let the following cut sets represent the failed systems cut sets for the accident sequence.

E1 * E2 * E3

E2 * E5 * E7

E1 * E2 * E5

Let the following fault tree represent the successful-systems fault tree.

TOP		OR	SYS1	SYS2	SYS3
SYS1	AND	E1	E6		
SYS2	AND	E1	E5		
SYS3	AND	E3	E4		

Since events E4 and E6 do not appear in the list of failed-systems cut sets, we can set them to house event FALSE and prune the fault tree, resulting in the following fault tree.

TOP		OR	~~SYS1~~	SYS2	~~SYS3~~
~~SYS1~~	~~AND~~	~~E1~~	~~FALSE~~		
SYS2	AND	E1	E5		
~~SYS3~~	~~AND~~	~~E3~~	~~FALSE~~		

Pruning this tree gives the following reduced fault tree.

TOP	AND	E1	E5

Solving this fault tree results in the following single cut set

E1 * E5

This cut set is used to reduce the failed-systems cut sets as follows.

E1 * E2 * E3
E2 * E5 * E7
~~E1 * E2 * E5~~

Whether specified externally by the user or internally by SAPHIRE, before the fault tree is solved, it is pruned depending on the structure of the tree and the house event setting. In order to do this, SAPHIRE again traverses the fault tree checking for house events. At each gate the algorithm checks each of the inputs to the gate to see if it has been set to any one of the three house event settings, "T," "F," or "I." If so then the logic for that gate is modified as follows. If the gate is an AND gate, then an input set to T or I is removed from the gate input list, while an input set to F causes the gate to be set to F. If the gate is an OR gate, then an input set to F or I is removed from the gate input list, while an input set to T causes the gate to be set to T.

The routine to check for house events and prune the logic of the fault tree is a recursive routine. Using the fault tree logic defined previously, along with the house event information and starting at the top gate in the fault tree, SAPHIRE checks each of the inputs to the current gate. If the input is a gate and the gate has not been previously checked, then the recursive routine calls itself to check this gate. The recursive routine returns a value of T, F, or I for each gate that is processed and it processes each gate only once. If a house event value is returned for the top gate, then there is no need to solve the fault tree and a message is displayed. If the value returned is T, the message "The TOP event has occurred (TRUE)!" will be displayed. If the value is F, then the message "The TOP event cannot occur (FALSE)!" will be displayed. If the value returned is I, then the message "No logic to solve!" will be displayed.

Coalescing Like Gates

The next step in the fault tree solution is to coalesce like gates. This process combines those gates that are input to other gates of the same type. Specifically, AND gates that are input to AND gates are combined and OR gates that are input to OR gates are combined. The following fault tree is an example of the coalescing of both an AND gate and an OR gate.

TOP	AND	GATE1	GATE2
GATE1	OR	GATE3	EVENT1
GATE2	AND	EVENT2	EVENT3
GATE3	OR	EVENT4	EVENT5

After coalescing, GATE2 is consumed by the TOP gate and GATE3 is combined with GATE1. The following fault tree is the result of these modifications.

TOP	AND	GATE1	EVENT2	EVENT3
GATE1	OR	EVENT1	EVENT4	EVENT5

In the above example, both gates that were coalesced were referenced only by gates of the same type. This resulted in the removal of both of these gates from the logic. The following example shows a case where the coalesced gate is not removed.

TOP	AND	GATE1	GATE2
GATE1	OR	GATE2	EVENT1
GATE2	AND	EVENT2	EVENT3

After coalescing, the following tree is generated:

TOP	AND	GATE1	EVENT2	EVENT3
GATE1	OR	GATE2	EVENT1	
GATE2	AND	EVENT2	EVENT3	

By coalescing the fault tree, the number of gates is reduced and the number of inputs to a gate is maximized. This process can substantially reduce the processing time as well as provide for better optimization later in the fault tree restructuring process. Note, however, that the total amount of space required to store the inputs to the fault tree can grow significantly as a result of coalescing the tree. The amount of additional space required depends on the number of gates that can be coalesced, the number of times a coalesced gate is referenced in the tree, and the number of inputs to the coalesced gate. This increased space requirement will usually be recovered during module and independent subtree processing later.

To perform the coalescing step, SAPHIRE starts with the TOP gate of the fault tree and recursively checks the list of inputs to the current gate. Any duplicate inputs in the list are removed. If the input is a gate and it is the same type as the current gate, then the list of inputs to this gate is added to the current gate input list. The gate reference is then removed from the list. If the input is a gate with a single input then the gate reference is replaced by its input. Once all inputs to all gates have been processed, then SAPHIRE makes a pass through the current gate list and eliminates any gates that are no longer needed due to any of the previous restructuring steps.

Modules versus Independent Subtrees

SAPHIRE uses two methods of optimization that are similar and should be clarified. These optimization methods are independent subtrees and modules. Before solving a fault tree, SAPHIRE converts all the logic into a logically equivalent form in terms of AND gates, OR gates, and basic events. The following discussion assumes this form of fault tree logic. In SAPHIRE, an *independent event* is defined as an event that is input to only one gate. An *independent gate* is a gate that is input to only one other gate and contains as inputs only independent events.

An *independent subtree* is a gate that has as inputs only independent events or independent gates. The inputs to an independent subtree can occur only once in a fault tree, however, an independent subtree may be input to many other gates. Note, the independence defined here is logical independence.

In SAPHIRE a set of events M={E1,E2,...,En} is defined to be a *module* of a fault tree if the following two conditions are met. (1) For every occurrence of E as input to a gate, the other events in M also occur as input to the same gate. (2) Every occurrence of M is an input to the same gate type, either an AND or an OR gate. These events can be combined under a single gate called a module. All references to these events are converted to reference the module. Once a module is created, all of the events input to it occur only as inputs to a single gate. Since a module may appear multiple times in a fault tree, it is usually not an independent gate, however, it is always an independent subtree. A gate that has a module as one of its inputs is only an independent subtree if the module is an independent gate.

In the fault tree reduction process, independent subtrees need not be expanded until the very end of the process. Once a fault tree is solved in terms of independent subtrees, it is a simple expansion process to convert the minimal cut sets to their basic event representation. Since a reduced number of tokens needs to be analyzed in the fault tree solution process, independent subtrees save large amounts of processing time. Figure 18 shows an example fault tree with a module and an independent subtree. In the example, Gate-3 also happens to be an independent gate.

Module Determination and Creation

The next step in the restructuring process is to find all modules in the fault tree. To perform this step, SAPHIRE uses a temporary bit vector. The bit vector contains one bit for each event in the fault tree. The first of these bit vectors keeps track of the events that are used in the fault tree. If complemented events are used, then a second bit vector is allocated for the complemented events.

A vector is also created for each gate currently defined. These vectors will contain, in bit format, the events used by each gate. We also define two vectors, TMP1 and TMP2, which hold intermediate results. Finally, we define an array containing one number for each event. This number is a count of the number of times each event is used in the fault tree.

Once the data arrays are created, we initialize the TMP1 vector and the event count array by traversing the input list. For each input, we check to see if it is an event, and if so, we set its bit in the TMP1 vector and increment the count for this event. If the event is complemented, then its bit is set in the complemented vector. When all inputs have been processed, we eliminate any event that occurs as both a complemented and a non-complemented event from the event vector list. These events cannot be included in modules. Next, we process each gate and set the appropriate bits in each gate's bit vector to reflect the events used by that gate. When this process is complete, we are ready to find the modules in the fault tree. Using the fault tree shown in Figure 18, the following initialized data structures would be defined.

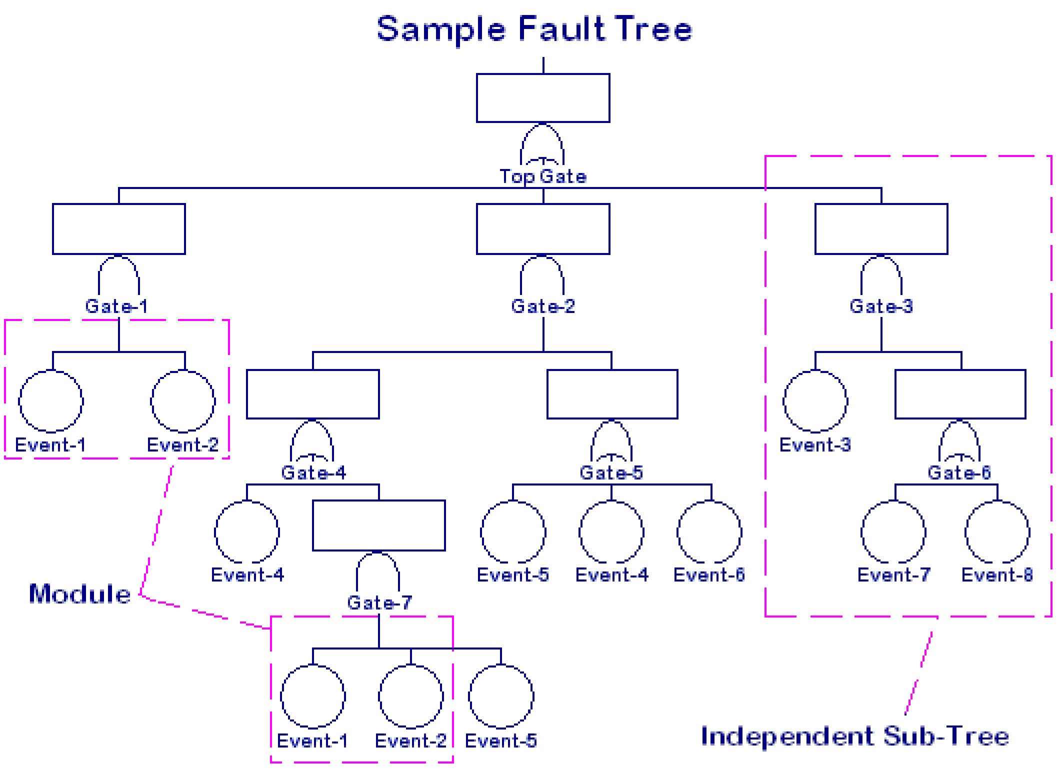


Figure 18. Independent subtree and module fault tree.

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
Used?	1	1	1	1	1	1	1	1

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TopGate								
Gate-1	1	1						
Gate-2								
Gate-3			1					
Gate-4				1				
Gate-5				1	1	1		
Gate-6							1	1
Gate-7	1	1			1			

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TMP1	1	1	1	1	1	1	1	1
TMP2								

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
Count	2	2	1	2	2	1	1	1

Using the TMP1 bit vector and the maximum number of events to be processed, we check to see if an event's bit is set. If the bit is set in the TMP1 vector for this event, then we look at all uses of this event to see if it occurs in combination with other events. We do this by initializing the TMP2 vector to the current list of events to process, TMP1. We then loop over the gate vectors checking to see if the current event is used by the gate. If it is used, then we perform a bit "AND" operation using the gate vector and the TMP2 vector. The result of the operation is stored in the TMP2 vector. We continue this process for each gate that uses the basic event. If at any time we find a gate that uses the event and is a different type than the other gates that use the event or the TMP2 vector has no events set, we exit the processing and continue with the next event. Using our data structures, the steps for Event-1 are as follows.

(1) Initialize TMP2 vector.

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TMP1	1	1	1	1	1	1	1	1
TMP2	1	1	1	1	1	1	1	1

(2) The first gate to use Event-1 is Gate-1, therefore, perform bit "AND' operation on Gate-1 and TMP2 storing results in TMP2.

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TMP1	1	1	1	1	1	1	1	1
TMP2	1	1						

(3) The next gate to use Event-1 is Gate-7, therefore, perform bit "AND" operation on Gate-7 and TMP2 storing results in TMP2.

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TMP1	1	1	1	1	1	1	1	1
TMP2	1	1						

No more gates use Event-1, therefore, the result of the above process is a bit vector, TMP2, containing those events that are always referenced together. We need to further check this list to ensure that none of these events are used elsewhere in the fault tree. We achieve this by checking the count of the number of times the event is referenced in the fault tree. If this count does not match the current event's count, then the event is removed from the list. In our example we see that Event-1 and Event-2 are in the TMP2 vector. Checking the count vector, we see that both events are used the same number of times (twice) in the fault tree.

If the remaining list is greater than one event, we create a new gate containing the events in the list and change all gates that reference the current event so they reference this new gate instead. The other events in the new gate are also deleted from any modified gate. Once this is done, we update our TMP1 vector containing the current list of events to process. This is done by complementing the TMP2 vector and performing a bit "AND" operation with the TMP1 vector. This effectively removes any events that we have put in a module from the list of events to be processed. In our example, we create a module using Event-1 and Event-2, then update the fault tree to use this module. The temporary bit vectors are updated as shown. Notice that both Event-1 and Event-2 are removed from the list of events to be processed.

	Event-1	Event-2	Event-3	Event-4	Event-5	Event-6	Event-7	Event-8
TMP1			1	1	1	1	1	1
TMP2	1	1						

The above operations continue until all events have been processed and no further restructuring is possible. When SAPHIRE has completed this step, one more loop through the tree is made to combine any gates that had all their inputs converted to a gate. This eliminates any single-input gates from the fault tree.

Independent Event Determination

The next step in the fault tree restructuring process is to determine which events are independent. For this purpose SAPHIRE defines "independent" as only occurring once in the fault tree. This step is performed by defining two bit vectors. Each time an event is encountered, a bit is set in the first vector. If the bit is already set, then the corresponding bit in the second vector is also set. When complete, the second bit vector represents the list of basic events that occur more than once. The events not in this list are independent.

Independent Gate and Subtree Determination

The next step in the restructuring of the fault tree is to determine the independent gates and subtrees in the fault tree. Independent subtrees are much easier to solve since they generate only minimal cut sets. SAPHIRE processes independent subtrees separately from the rest of the fault tree.

To find the independent gates and subtrees, SAPHIRE again uses a recursive routine to traverse the fault tree. SAPHIRE uses the data structures defined previously to check the inputs to each gate. If all the inputs to the gate are independent events and the gate occurs only once, then it is marked as an independent gate. If the input is a gate and has not been processed, then the routine calls itself to check this gate. If all inputs to the gate are independent events or gates, then the gate is flagged as an independent subtree. This results in a fault tree that has all independent subtrees identified.

Determining Gate Levels

The last step in the fault tree restructuring process is to determine the gate levels. The TOP gate is defined to have level 0. Its inputs have level 1, the inputs to those gates have level 2, and so forth. The *level* of a gate is the number of gates one encounters after the TOP in going from the TOP to the gate of interest. If a gate appears more than once in a tree, define the gate's level as the largest of the levels corresponding to the various places where the gate occurs. To determine the level of each gate, a recursive routine is used. This routine keeps track of the level for each gate. Each time the gate is encountered in the traversal of the fault tree, its level is checked against the current level. If the current level is greater than the gate's assigned level, then the gate's level is set to the current level. The routine exits early if a gate's level is greater than or equal to the current level. This process continues until the entire tree has been processed.

This information is used later in determining the *expansion path* for the fault tree. The expansion path for a fault tree is the order in which the gates for a fault tree are solved. This expansion path can significantly affect the time it takes to solve a fault tree. SAPHIRE attempts to determine the optimal expansion path.

Fault Tree Reduction

Once the fault tree is loaded and restructured, it is ready to be solved. This process consists of a number of steps that convert the Boolean logic representing the fault tree to its expanded form representing the desired minimal cut sets for the tree. In SAPHIRE, a fault tree may represent either a system equation or a sequence equation. In either case, the same algorithm is used to solve the tree.

Cut Set Truncation

The exact solution of many large fault trees can prove to be prohibitive; therefore, various methods have been developed to reduce the time required to solve a fault tree. SAPHIRE allows the user to specify that a number of these methods be used in the fault tree solution. The first and most common method is to eliminate any cut set whose probability falls below a specified truncation value. The second method is to eliminate any cut set that has more than a specified number of unique events in it. The third method is to eliminate any cut set that has more than a specified number of zone flagged events in it. A *zone flagged event* is an event that has been marked as representing a zone (location or area). In a facility, a fire zone may represent a room with fire barriers around it. A security zone may represent an area with certain security characteristics. This method is used in location analysis to allow for the truncation on the number of zone events in a cut set. The last method provided in SAPHIRE for cut set truncation is typically used in seismic analysis and allows the user to combine the first truncation method with another criterion that checks to see if any event in the cut set is below a specified probability before it is truncated.

All of the above truncation methods are supported by SAPHIRE. The user may also choose to solve the fault tree exactly. No matter which methods are used, SAPHIRE attempts to take advantage of whatever it can to simplify and reduce the amount of work required to solve a tree. The ways each of these truncation methods is implemented will be discussed in detail as the process for the fault tree solution is described.

Intermediate Result Caching

Fault tree solutions can easily generate enough intermediate cut sets to fill up all available computer memory. Therefore, a method is required to allow this data to be written out to a secondary data storage area. SAPHIRE uses a disk caching technique to store the intermediate data. This allows for the processing of large amounts of intermediate data. The limit is the amount of available disk space on the computer being used. This also allows SAPHIRE to be run on a minimal computer without memory beyond the 640K available to standard DOS applications. SAPHIRE does, however, allow the user with a more powerful computer and additional extended memory to create a virtual disk and direct the intermediate information that would have resided on the hard disk to the virtual disk. This will improve the performance of SAPHIRE on large problems by a factor of 3 to 5 times. This overview will not attempt to describe in detail how the cache software works. The performance of any fault tree reduction software is quite dependent on the methods used to handle the large amounts of intermediate data; therefore, the user should ensure that an efficient method is used.

Fault Tree Cache Initialization

The first step in the fault tree reduction process is to take the fault tree logic that has been loaded and restructured and store this logic in a format for efficient use and retrieval by the fault tree reduction software. This process includes the creation and initialization of certain data structures containing information that is used during the solution process to simplify and speed up the fault tree reduction process. By including this data in a data structure and updating it as the fault tree is solved, SAPHIRE is able to avoid many additional calculations.

Using the gate level information determined previously, SAPHIRE creates an ordered table such that all gates for a given level appear before any gates for the next larger level. Any independent subtrees appear after all nonindependent gates for the fault tree. This ordering defines the expansion path to be used for solving the fault tree. As mentioned previously, the SAPHIRE algorithm is essentially a top-down approach, but strictly speaking, the algorithm processes the fault tree first from the bottom up, then from the top down. The algorithm is bottom up because we treat each OR gate as a mini fault tree and solve them starting with the last gate or the bottom of the fault tree. When all OR gates up to the TOP gate have been solved, SAPHIRE expands the TOP gate from the top down.

As the fault tree logic table is being created, SAPHIRE generates some information to be used during the expansion process to help in cut set truncation. A bound can be calculated on the contribution of the independent subtrees to the cut set probabilities. If the user has specified truncation on probability, this bound can be used to eliminate cut sets earlier than otherwise possible. For now, let BPC denote this Bound on the Probability Contribution. Calculate the BPC for any gate as follows. The BPC for a basic event is its probability. The BPC for an

AND gate is the product of the BPC's of the inputs. The BPC for an OR gate is the largest BPC of the inputs. Since the gate table is ordered by level, these calculations can be performed one gate at a time, starting with the last gate and proceeding to the top of each independent subtree.

To see how this works, suppose first that *S* is an independent subtree with only two inputs, *A* and *B*, both basic events. Because *S* is independent, as defined in Sections 5.9 and 5.12, each of its basic events appears only once, so *A* and *B* do not appear in any other part of the fault tree. Because basic events are assumed to be independent in the statistical sense of Section 4.6, *A* and *B* are statistically independent of each other and of the rest of the tree.

Any cut set that *S* contributes to will have the form (*S* AND other terms). If *S* is an AND gate, this form is (*A* AND *B* AND other terms), and the probability of the cut set is *P*(*A*)*P*(*B*)*P*(other terms), by independence. This equals BPC(*S*)H*P*(other terms), by the definition of BPC for an AND gate. If instead *S* is an OR gate, any cut set that *S* contributes to will have the form (*A* and other terms) or else (*B* and other terms). The cut set probabilities are bounded by

max[*P*(*A*), *P*(*B*)]x*P*(other terms)

which equals BPC(*S*)x*P*(other terms), by the definition of BPC for an OR gate.

In either case, any cut set that *S* contributes to has probability bounded by the value of BPC for *S*. The same idea is true if *S* has more than two inputs, and if they are not necessarily basic events but may be independent gates instead. Therefore, if BPC for *S* is less than the truncation value, *S* can be eliminated from the tree. In any case, the BPC is calculated and stored so that it can be used to eliminate cut sets earlier than otherwise possible.

If the user has chosen to truncate on size or zones a similar calculation can be performed on independent subtrees to get a size contribution of the subtree to each cut set it appears in. If size truncation is selected, then all basic events are counted. If zone truncation is selected, then only events that are zone flagged are counted. At each AND gate, the size contributions of the inputs are added together. For a qualified basic event the size is one. For a gate, however, the size may be larger than one. At each OR gate, the size contribution of the smallest input is used as the size contribution of the gate. Once these values are calculated, they are stored in the gate table for future use. The fault tree is now ready to be expanded.

Fault Tree Gate Expansion

The process of solving a fault tree involves three basic steps. These steps are gate expansion, Boolean absorption, and cut set truncation. In the first step, the gates of the fault tree are expanded by replacing them with their inputs. In the second step, the first four of the following identities are applied to the cut sets:

(1) $A * A = A$

(2) $A + A * B = A$

(3) $A * B * /A = \text{i}$

(4) $//A = A$

(5) $A * B + A * /B = A$ (not currently applied).

The first identity (idempotent relationship) prevents two identical events from appearing in the same cut set. The second one (absorption relationship) is the most computationally difficult to apply. In terms of set theory it consists of eliminating subsets, because $A*B$ is a subset of A. Computer programmers, on the other hand, tend to think of the identity as eliminating supersets; $A*B$ is regarded as a larger entity than A because it has more tokens to manipulate. Both the subset and superset terminology can be found in the literature, but this document will use only the term "absorption." The absorption identity is used to eliminate cut sets that are not minimal. The basis for using the Law of Absorption is that the top gate has become a giant OR gate with the cut sets as inputs. If A and A*B are cut sets, the top gate contains A + A*B, which can be simplified to A. The third identity (exclusion relationship) implies that no cut set will contain both the failure and the success of an event. The fourth identity (double negation relationship) states that the complement of a complemented event is the event itself. Identity number five (exhaustion relationship) is not currently performed by SAPHIRE. It is important to note that SAPHIRE does not currently calculate prime implicants (Quine 1959). Complemented events appear in the cut sets with a "/" in front of the event name.

The final step, cut set truncation, involves the elimination of cut sets that fall outside user specified truncation limits. There have been many different methods applied to performing these three steps. Some codes use a top-down approach, while others use a bottom-up approach. Both approaches have their strong points. SAPHIRE uses some features from each approach to optimize the fault tree solution process.

Using the fault tree logic definition generated previously, SAPHIRE begins expanding the tree. Since OR gates increase the number of cut sets, the algorithm treats all OR gates in the fault tree as mini fault trees. These trees are solved first, starting with the last nonindependent OR gate and proceeding to the TOP gate of the fault tree. All absorption and truncation techniques are applied on these small trees, eliminating cut sets as soon as possible. When the TOP gate is encountered, it is solved using as input all the cut sets generated by solving the mini fault trees described above. The result of this approach is to partition the large fault tree into many smaller subtrees that are easier to solve. The fewer cut sets generated for the smaller trees will also tend to require less time to apply the absorption identities and to truncate.

Note that the cut sets generated by the above process are in terms of independent subtrees. When the TOP gate has been solved and all absorption has been performed, the independent subtrees are expanded. This step requires no absorption; independent subtrees can only generate cut sets that are minimal.

Cut Set Absorption

As the fault tree expansion occurs, cut sets are checked at each gate to see if they can be eliminated. There are several ways a cut set may be eliminated during the expansion process.

SAPHIRE maintains the current bound on the probability contribution (BPC defined in Section 5.17) and size for each cut set throughout the fault tree expansion. These contributions are updated depending on the type of expansion being performed. By keeping current BPC values, SAPHIRE does not need to recalculate these values each time the cut set is modified or expanded. Much computation time is saved by this approach.

If the gate to be expanded is an OR gate, then SAPHIRE also compares the inputs to the OR gate against the inputs of the cut set containing the OR gate. If there is a common event, then the reference to the OR gate can be removed and the cut set need not be expanded further. The reason for this is that any cut sets generated from an OR gate of this type will be absorbed later in the process anyway. The following example demonstrates this process.

The cut set

GATE1 * EVENT1 * EVENT2

and the following definition of GATE1 as an OR gate with three inputs

GATE1 OR EVENT1 EVENT3 EVENT4

will generate the following cut sets when expanded.

EVENT1 * EVENT2
EVENT1 * EVENT2 * EVENT3
EVENT1 * EVENT2 * EVENT4

Notice that the second and third cut sets are absorbed by the first.

Boolean Absorption

The process of performing the Boolean absorption reduction can be a time-consuming operation. The methods used in SAPHIRE are described in Corynen (1988). This method uses a set of bit tables to determine those cut sets that can be absorbed by a given cut set. For a detailed description of the process, refer to the indicated document. This method is very powerful and has good run-time characteristics. In order to be most effective with this algorithm or any other one used for the Boolean absorption process, the number of cut sets compared must be minimized. The expansion approach described previously tends to generate smaller numbers of intermediate cut sets, minimizing the amount of time spent on absorption.

Data Storage Considerations

Given the task to be performed in solving a fault tree, an optimal format for storage and retrieval of the intermediate cut set data must be determined. Two obvious methods were considered in SAPHIRE. First, since a large amount of time can be spent in the determination of sets to be absorbed, one option is to store the intermediate data in a format that can be directly used by the absorption routine. This format would be an array of bit vectors with each row of the array representing an event and each column representing a cut set. This format was used in the first version of SAPHIRE and worked well for small problems because the bit vector arrays could be easily contained in the computer's fast memory. As problem size increased and it became necessary to shift these arrays to disk, this method of storage became difficult to manage efficiently.

The second alternative is to store the cut sets as an array of numbers representing the events in each cut set. The first number is a count representing the number of events in the cut set. This number would be followed by a probability value, a size value, and a list of numbers representing the gates or events contained in the cut set. The list is terminated by a zero count number. This format is the one used in the current version of SAPHIRE. It is simple and easy to store and retrieve from intermediate storage. The process of gate expansion is also easily handled with this format. When absorption is performed, SAPHIRE creates the array of bit vectors. As problem size increases, this format has proven to be much more flexible and easy to manage than the first.

Sequence Cut Set Generation

Another area that must be considered when developing a risk assessment code is the accident sequence analysis. Accident sequences are defined in SAPHIRE by developing event trees. SAPHIRE provides a graphical editor to use in developing event trees. Figure 19 shows an example of an event tree developed in SAPHIRE. Once the user has developed the event tree, SAPHIRE automatically generates the sequence logic from the graphical event tree. The sequence logic is the list of systems that succeed or fail during this accident sequence. These system failures and successes are top events of fault trees. This logic is used by SAPHIRE to generate the cut sets for the sequence.

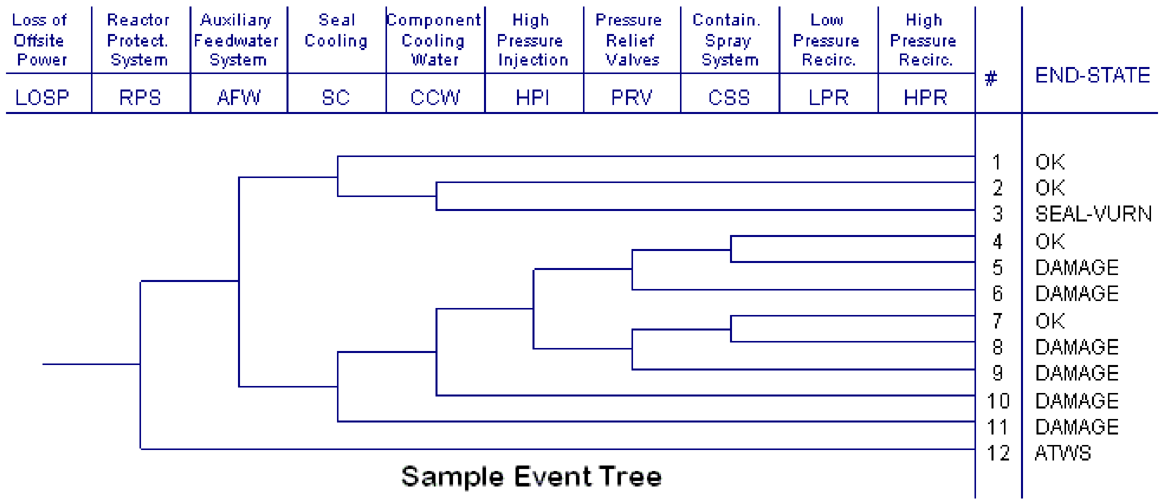


Figure 19. SAPHIRE event tree.

There are two methods that can be used to generate sequence cut sets. First, the cut sets generated by solving the system fault trees can be used as input to the accident sequence algorithm. This method simply combines the cut sets for each system as defined by the sequence logic. The second method is to create a fault tree for a sequence by combining the fault trees corresponding to system failures and successes for the sequence. The fault tree reduction algorithms can then be used to solve the accident sequence. SAPHIRE allows the user to select either method, but only the latter method will be discussed here.

In SAPHIRE, accident sequences are defined using an event tree to indicate the failure or success of top events. Each top event in the event tree is associated with a system fault tree. To solve the accident sequence, SAPHIRE constructs a fault tree for those systems that are defined to be failed in the sequence logic by creating a dummy AND gate with these systems as inputs. In Figure 19, the accident sequence logic for sequence 9 is

LOSP * /RPS * AFW * /HPI * /PRV * CCS * LPR

Therefore, SAPHIRE creates the following failed systems fault tree

FAILED	AND	AFW	CCS	LPR
AFW	TRAN			
CCS	TRAN			
LPR	TRAN			

where AFW, CCS, and LPR represent the fault tree logic for Auxiliary Feedwater System, Containment Spray System, and Low Pressure Recirculation system, respectively, and TRAN denotes a transfer to the system fault tree.

SAPHIRE then solves this fault tree using the specified truncation values. This process results in a list of cut sets for the failed systems in the accident sequence. SAPHIRE then uses the "cut set matching" technique to further reduce this list of failed-system cut sets. This technique uses the cut sets determined from solving the successful-system fault trees in the accident sequence logic to eliminate cut sets from the list of failed-system cut sets. To do this, SAPHIRE first scans the list of failed-system cut sets and assigns a value of FALSE to any basic event that does not appear in this list. Once this is done, the fault tree representing the successful systems in the accident sequence logic is constructed, pruned by the house events, and solved. The successful systems fault tree for accident sequence 9 is

SUCCESS	OR	RPS	HPI	PRV
RPS	TRAN			
HPI	TRAN			
PRV	TRAN			

where RPS, HPI, and PRV represent the fault tree logic for the Reactor Protection System, High Pressure Injection system, and the Pressure Relief Valves, respectively. This fault tree models failure of the RPS system, the HPI system, or the PRV system. The top event of the tree does not occur as part of accident sequence 9. That is, none of the cut sets in the tree occur.

The minimal cut sets for the sequence remain after the successful-system cut sets are deleted. There are a couple of points to note in this process. First, each sequence has an initiating event frequency associated with it. If the user specifies a probability truncation value, SAPHIRE divides this value by the initiating event frequency. This eliminates the need to handle the initiating event during the fault tree reduction phase. Second, during the processing of an accident sequence, certain pieces of equipment or trains of a system may need to be either failed or ignored. SAPHIRE allows the user to specify a set of house event flags to be associated with a particular sequence. These flags allow the user to automatically prune the fault tree logic before it is solved by setting basic events to house events and reducing as described in Section 5.7. The result is a fault tree with the specified components in the specific state required by the sequence.

QUANTIFICATION TOOLS FOR PROBABILITIES AND FREQUENCIES

This section provides an overview of fault tree and accident sequence quantification using minimal cut sets. Vesely et al. (1981) and Fussell (1975) contain additional details and references for the interested reader. The section is written in terms of failure probabilities, but is also correct if the term "probability" or "failure probability" is replaced everywhere by "unavailability."

Quantifying Minimal Cut Sets

The individual cut set probabilities are determined by multiplying the probabilities of the applicable basic events.

$$C_i = q_1 q_2 \cdots q_n \qquad (6\text{-}1)$$

where

Ci	=	probability of cut set *i*, and
qk	=	probability of the *k*-th basic event in the ith cut set.

This follows from Equation (4-8) and the assumed statistical independence of the basic events.

Quantifying Fault Trees

The fault tree quantification process is performed in two steps: (1) calculation of individual cut set probabilities, which was described above in Section 6.1, and (2) combining the cut set probabilities. The exact probability of the union of the cut sets can be found, in principle, by Equation (4-6), where each *Ai* is a cut set. This is normally much too cumbersome. Therefore, two approximations are often used, the rare event approximation and the minimal cut set upper bound. Examples are calculated in Sections A4 and A5 of Appendix A.

Rare Event Approximation

A common approach to calculate the probability for a top event is to add together the probabilities for the cut sets, where the cut set probability is given by Equation (6-1). Thus, the rare event approximation is

$$S = \sum_{i=1}^{m} C_i \qquad (6\text{-}2)$$

This approximation is a good approximation when the cut set probabilities are small. In screening analyses, when relatively large screening values are used to bound the component failure probabilities, the rare event approximation can exceed 1.

Minimal Cut Set Upper Bound

The minimal cut set upper bound calculation is an approximation to the probability of the union of the minimal cut sets for the fault tree. The equation for the minimal cut set upper bound is

$$S = 1 - \prod_{i=1}^{m} (1 - C_i) \qquad (6\text{-}3)$$

where

S = minimal cut set upper bound for the system unavailability,

Ci = probability of the ith cut set, and

m = number of minimal cut sets in the fault tree.

The minimal cut set upper bound is always less than or equal to 1. The input values for the minimal cut set upper bound are probabilities. Barlow and Proschan (1981) show that Equation (6-3) gives an upper bound on the exact probability of the top event.

The minimal cut set upper bound works well with fault trees containing only AND and OR gates without complemented events or NOT gates. With noncoherent fault trees, that is, trees that contain NOT gates and/or complemented events, the minimal cut set upper bound can produce results that are overly conservative. The magnitude of the overestimation will depend upon the structure of the tree. In such cases, other calculational techniques should be used such as the SIGPI algorithm (Patenaude 1987). In most cases, the minimal cut set upper bound will produce reliable results.

Warning: When *C*i is very small (on the order of 1E-15), 1 - *C*i is rounded off to 1.0. If this happens for most or all of the *C*i's, the product in Equation (6-3) will be too large, and the bound *S* will be too small. Although *S* is an upper bound in theory, in practice it is not computed to sufficient accuracy when the *C*i's are extremely small. In such a case the rare event approximation, given by Equation (6-2), is better.

Quantifying Sequences

An accident sequence begins with an initiating event, which has a frequency *f*. The units of the frequency are 1/time, and there is no theoretical upper bound on its possible value. This distinguishes a frequency from a probability, which is unitless and bounded by 1.0.

After the initiating event, various systems in the plant are suppose to function in sequence. Depending on whether they function or not, the sequence can proceed to different possible plant states. Consider one of these systems. Given the assumed initiating event and the success or failure of the systems that were invoked earlier in the sequence, the probability of the system's failure is quantified by a fault tree for the system. For each such sequence of interest, SAPHIRE constructs and simplifies the fault tree for the entire sequence, by combining the fault trees for the failed systems and the negation of the fault trees for the successful systems, as described in Section 5.22.

Let *S* be the probability of the sequence fault tree, evaluated using the minimal cut set upper bound or the rare event approximation. It is a probability calculated assuming that the initiating event has occurred. Then, the frequency of the sequence is the product *fS*. In this way, sequence frequencies are found.

EVENT PROBABILITY CALCULATION TYPES

The calculation type specifies the method to be used to calculate the basic event probability. Thirteen types are available in SAPHIRE, and they are summarized in Table 1. The resulting probability for Types 1 through 7 will be the mean used in the uncertainty analysis described in Section 9. Types 2, 4, and 6 are approximations of the exact formulas given by Types 3, 5, and 7.

Table 1 - SAPHIRE Calculation Types

Type	Calculation Method
1	Probability
2	Lambda * Mission Time
3	1 - Exp(-Lambda * Mission Time)
4	Lambda * Min(Mission Time, Tau)
5	Operating Component with Repair (Full Eq)
6	Lambda * Tau / 2.0
7	1 + (Exp(-Lambda*Tau)-1.0) / (Lambda * Tau)
T	Set to House Event (Failed, Prob=1.0)
F	Set to House Event (Successful, Prob=0.0)
I	Ignore this Event (Remove it from logic)
S	Set to System Min Cut Upper Bound
E	Set to End State Min Cut Upper Bound
G	Enter Ground Acceleration for Screening
H	Use Hazard Curve for screening G-Level

Calculation Type 1

Probability

Calculation Type 1 takes the number specified by the user in the Probability field as the basic event failure probability. This is the type used for demand probabilities.

Calculation Type 2

Lambda * Mission Time

Calculation Type 2 uses the number provided for λ as the basic event failure rate per hour and multiplies it by the basic event mission time expressed in hours.

If the basic event mission type, expressed in hours, is not input then the global or system mission time is used. The global mission time is set by the user in the **Utility | Define Constants** option. A default mission time of 24 hours is provided by SAPHIRE until it is changed by the user.

This calculation is the rare event approximation to the actual failure probability for an operating component without repair during the mission time. This approximation is relatively good for failure probabilities less than 0.1.

Calculation Type 3

1 - Exp(-Lambda * Mission Time)

Calculation Type 3 uses the actual equation for failure probability for an operating component without repair,

$$q = 1 - e^{-\lambda t}$$

where

q = failure probability of the basic event,

λ = failure rate per hour, input as λ, and

t = mission time expressed in hours.

Calculation Type 4

Lambda * Min(Mission Time, Tau)

Calculation Type 4 is a rare event approximation for the failure of an operating component with repair. The approximation is λ times τ. It uses λ as the per hour failure rate and τ as a user-specified time to repair in hours. If the mission time t is less than τ, then $\lambda * t$ is a better approximation of the event probability; therefore, SAPHIRE uses λ times the minimum of τ and mission time.

Calculation Type 5

Operating Component with Repair (Full Eq)

Calculation Type 5 is the actual equation for the failure probability of an operating component with repair. The equation is

$$q = \frac{\lambda\tau}{1+\lambda\tau}\left(1 - e^{-(\lambda + \frac{1}{\tau})t}\right)$$

where

q = failure probability of the basic event,

λ= failure rate per hour, input as λ,

t = mission time expressed in hours, input as a default, and

τ = average time to repair expressed in hours, input as τ.

Calculation Type 6

Lambda * Tau / 2.0

Calculation Type 6 is the rare event approximation for the failure probability of a standby component with a surveillance test interval. The equation used is

$$q = \frac{\lambda T}{2}$$

where

q = failure probability of the basic event,

λ= standby failure rate per hour, input as λ, and

T = surveillance test interval in hours, input as τ.

Calculation Type 7

1 + (Exp(-Lambda*Tau)-1.0) / (Lambda * Tau)

Calculation Type 7 is the actual equation for the failure probability of a standby component with a surveillance test interval. The equation is

$$q = 1 + \frac{e^{-\lambda T} - 1}{\lambda T}$$

where

q = failure probability of the basic event,

λ= standby failure rate per hour, input as λ, and

T = surveillance test interval in hours, input as τ.

Calculation Types T, F, and I

Calculation Types T, F, and I are used to set basic events to house events.

Set to House Event (Failed, Prob=1.0)

Calculation Type T turns the basic event into a house event that always occurs (probability 1.0).

Set to House Event (Successful, Prob=0.0)

Calculation Type F turns the basic event into a house event that never occurs (probability 0.0). If the event states that a component fails, T forces the component to fail while F forces it to succeed.

Ignore this Event (Remove it from logic)

Calculation Type I indicates that the basic event is to be treated as if it did not exist in the logic for the fault tree.

Setting an event to a house event actually changes the logic of the fault tree, pruning appropriate branches and events from the fault tree. Therefore, the flags on the affected fault trees will indicate a need to generate new cut sets rather than just requantifying existing cut sets.

SEE ALSO

House Event Pruning for details on the processing of house events

Calculation Type S

Set to System Min Cut Upper Bound

Calculation Type S indicates that the probability of the basic event is to be determined by finding a system with the same name as the basic event. Then, use the minimal cut set upper bound for this system as the failure probability for the basic event.

SAPHIRE will accept numbers in scientific or decimal format. For example, 1.E-4 and 0.0001 are both valid inputs.

NOTE: When using the short-hand scientific notation, a decimal point must precede the "E", thus 1E-2 will not be accepted but 1.E-2 or 1.0E-2 will. SAPHIRE will accept an uppercase E or a lowercase e. Also, note that 1.0E-020 is not the same as 1.0E-02. This has caused confusion in the past.

Calculation Type E

Set to End State Min Cut Upper Bound

Calculation Type E indicates that the probability of the basic event is to be determined by finding an end state with the same name as the basic event. Then, use the minimal cut set upper bound for this end state as the failure probability for the basic event.

SAPHIRE will accept numbers in scientific or decimal format. For example, 1.E-4 and 0.0001 are both valid inputs.

NOTE: When using the short-hand scientific notation, a decimal point must precede the "E", thus 1E-2 will not be accepted but 1.E-2 or 1.0E-2 will. SAPHIRE will accept an uppercase E or a lowercase e. Also, note that 1.0E-020 is not the same as 1.0E-02. This has caused confusion in the past.

Calculation Type G

Enter Ground Acceleration for Screening

Calculation Type G indicates that the basic event is to be treated as a seismic event. The probability value will be the ground acceleration to be used to determine the probability of failure given the event's killer acceleration, beta random, and beta uncertainty, or given the event's histogram number.

Calculation Type H

Use Hazard Curve for screening G-Level

Calculation Type H indicates that the point probability for this event is to be determined from the Hazard Curve Histogram for this family.

IMPORTANCE MEASURES

Types of Importance Measures

SAPHIRE calculates seven different basic event importance measures. These are the Fussell-Vesely importance, risk reduction ratio, risk increase ratio, Birnbaum or first derivative importance, risk reduction difference, risk increase difference, and the structural importance. These importance measures are calculated for each basic event for the respective fault tree or accident sequence.

The ratio importance measures are dimensionless and consider only relative changes. The difference definitions account for the actual risk levels that exist and are more appropriate when actual risk levels are of concern, such as comparisons or prioritizations across different plants. For purely relative evaluations, such as prioritizations within a plant, the ratios sometime give more graphic results.

The main importance measures are

- FussellVesely importance, an indication of the percentage of the minimal cut set upper bound contributed by the cut sets containing the basic event

- Risk reduction , an indication of how much the minimal cut set upper bound would decrease if the basic event never occurred (typically, if the corresponding component never failed)

- Risk increase , an indication of how much the minimal cut set upper bound would go up if the basic event always occurred (typically, if the corresponding component always failed)

- Structural importance , the number of cut sets that contain the basic event.

In SAPHIRE, the Basic Event Importance display lists the basic event name, its failure probability, the number of cut sets in which the basic event occurs, and three of the six importance measures. The user can choose to display either ratios or differences by setting a user constant. If the user selects ratios then the Fussell-Vesely importance, risk reduction ratio, and risk increase ratio are displayed together. Otherwise, the Birnbaum importance, risk reduction difference, and risk increase difference are displayed together. The list can be sorted on any column in the display.

The exposition below is written in terms of fault trees and event probabilities. However, SAPHIRE also can calculate importances for events in sequences. Recall that a sequence is simply a fault tree preceded by an initiating event with frequency *f*, where *f* has units 1/time. The frequency of any event in the fault tree is *f* times the probability of the event. Therefore, the ratio importances are unchanged whether the event is part of a fault tree or a sequence. A difference importance for an event in a sequence is *f* times the importance of the event in the fault tree. The maximum possible value of a difference importance is 1.0 if the event is in a fault tree and *f* if the event is in a sequence. This alternative formulation is indicated below by phrases in parentheses.

SEE ALSO

Calculational Details

Calculational Details

This section contains the calculational definition of the importance measures. Examples are given in Section A6 of Appendix A. Both the ratio and the difference are discussed in the appropriate sections. For the basic event under consideration, several notations are used repeatedly.

F(x)	=	minimal cut set upper bound (sequence frequency) evaluated with the basic event probability at its mean value.
F(0)	=	minimal cut set upper bound (sequence frequency) evaluated with the basic event probability set to zero.
F(1)	=	minimal cut set upper bound (sequence frequency) evaluated with the basic event failure probability set to 1.0.

Fussell-Vesely Importance

The Fussell-Vesely importance is an indication of the fraction of the minimal cut set upper bound (or sequence frequency) that involves the cut sets containing the basic event of concern. It is calculated by finding the minimal cut set upper bound of those cut sets containing the basic event of concern and dividing it by the minimal cut set upper bound of the top event (or of the sequence). In SAPHIRE, this calculation is performed by determining the minimal cut set upper bound (sequence frequency) with the basic event failure probability at its mean value and again with the basic event failure probability set to zero. The difference between these two results is divided by the base minimal cut set upper bound to obtain the Fussell-Vesely importance. In equation form, the Fussell-Vesely importance FV is

$$FV = [F(x) - F(0)]/F(x) \quad .$$

Risk Reduction

The risk reduction importance measure is an indication of how much the results would be reduced if the specific event probability equaled zero, normally corresponding to a totally reliable piece of equipment. The risk reduction ratio is determined by evaluating the fault tree minimal cut set upper bound (or the sequence frequency) with the basic event probability set to its true value and dividing it by the minimal cut set upper bound (sequence frequency) calculated with the basic event probability set to zero. In equation form, the risk reduction ratio RRR is

$$RRR = F(x)/F(0) \quad .$$

The risk reduction difference indicates the same characteristic as the ratio, but it reflects the actual minimal cut set upper bound (sequence frequency) levels instead of a ratio. This is the amount by which the failure probability or sequence frequency would be reduced if the basic event never failed.

The risk reduction difference (RRD) is calculated by taking the difference between the mean value and the function evaluated at 0. In equation form, the risk reduction difference RRD is

$$RRD = F(x) - F(0) \; .$$

Risk Increase

The risk increase ratio is an indication of how much the top event probability (frequency) would go up if the specific event had probability equal to 1.0, normally corresponding to totally unreliable equipment. The risk increase ratio is determined by evaluating the minimal cut set upper bound (sequence frequency) with the basic event probability set to 1.0 and dividing it by the minimal cut set upper bound evaluated with the basic event probability set to its true value. In equation form, the risk increase ratio RIR is

$$R/R = F(1)/F(x)\ .$$

The risk increase difference RID is calculated by taking the difference between the function evaluated at 1.0 and the nominal value. In equation form, the risk increase difference RID is

$$RID = F(1) - F(x)\quad .$$

Birnbaum Importance

The Birnbaum importance measure is calculated in place of the FussellVesely importance measure when differences are selected instead of ratios. The Birnbaum importance is an indication of the sensitivity of the minimal cut set upper bound (or sequence frequency) with respect to the basic event of concern. It is calculated by determining the minimal cut set upper bound (or sequence frequency) with the basic event probability of concern set to 1.0 and again with the basic event probability set to 0.0. The difference between these two values is the Birnbaum importance. In equation form, the Birnbaum importance *B* is

$$B = F(1) - F(0)\quad .$$

The Birnbaum importance can be interpreted as follows as an approximation of a derivative. If basic event *i* has probability *pi*, the rare-event approximation says that the top event probability, *F*, can be written as

F ≐ sum of cut set probabilities
= sum of probabilities of cut sets involving event *i* + sum of other cut set probabilities

The cut sets that involve event *i* all have probabilities of the form

pi@(product of probabilities of other basic events).

It follows that

$$F \doteq p_i \bullet A + C \quad ,$$

where *A* and *C* are sums of products of basic event probabilities that do not involve *pi*. Therefore, *F* is approximately a linear function of *pi*, and therefore,

∂F/∂*pi*, is approximately equal to [*F*(1) - *F*(0)]/(1 - 0), which equals the Birnbaum importance.

Uncertainty Importance

The incertainty importance of basic event *i* is defined by Iman and Shortencarrier (1986) as *σi*·∂F/∂*pi*, where *pi* is the probability of event *i* and *Fi* is the standard deviation of *pi*, reflecting uncertainty in *pi*. SAPHIRE calculates the uncertainty importance as *FiBi*, where *Bi* is the

Birnbaum importance of event *i*. By the above discussion of the Birnbaum importance, the SAPHIRE calculation uses the rare-event approximation of the derivative.

The definition of the uncertainty importance is motivated as follows. (See Iman and Shortencarrier for a somewhat different motivation.) The first-order Taylor-series approximation of the top-event probability, *F*, is

$$F \doteq F_{nom} + \sum_i (p_i - p_{i,}nom)\delta F/\delta p_i \quad ,$$

where the subscript *nom* denotes the nominal value. The derivatives are all evaluated at the nominal values. Now let each *pi* have an uncertainty distribution, with variance *Fi*2. It follows that the variance
of *F* is

$$var(F) \doteq \sum_i \sigma_i^2 (\delta F/\delta p_i)^2 \quad .$$

The quantity *σi*2(∂*F*/∂*pi*)2 is the contribution of event *i* to the total variance of *F*, and is the reduction in variance that would result from perfect knowledge (zero variance) of *pi*. The square root of this quantity is defined as the uncertainty importance.

UNCERTAINTY AND MONTE CARLO

The uncertainty analysis allows the user to calculate the uncertainty in the top event probability resulting from uncertainties in the basic event probabilities. To use this option, the user must have previously loaded or generated the cut sets and loaded the component reliability information and distribution data. Bohn et al. (1988) contains an excellent discussion of uncertainty analysis. A very brief overview is given here, with elaborations in the subsequent sections.

In an uncertainty analysis, SAPHIRE already has the top event expressed in terms of minimal cut sets, either generated earlier or loaded from some other source. These cut sets depend on many basic events, each of which has a probability described in terms of some parameter(s). For definiteness in this explanation, suppose that a basic event probability depends on the parameter λ. The value of λ for each basic event is not known exactly, but is estimated based on data or on expert opinion. The uncertainty in λ is quantified by a probability distribution: the mean of the distribution is the best estimate of λ, and the dispersion of the distribution measures the uncertainty in λ, with a large or small dispersion reflecting large or small uncertainty, respectively, in the true value of λ. This distribution is the *uncertainty distribution* of λ.

The computer programming treats an initiating event of a sequence as a basic event. It differs from the other basic events in only two ways: it has a frequency instead of a probability, and every cut set contains exactly one initiating event.

For all the basic events, SAPHIRE randomly samples the parameters from their uncertainty distributions, and uses these parameter values to calculate the probability of the top event. This sampling and calculation are repeated many times, and the uncertainty distribution for the probability of the top event is thus found empirically. The mean of the distribution is the best estimate of the probability of the top event, and the dispersion quantifies the uncertainty in this probability. For an accident sequence the process is the same, except the sequence fault tree is preceded by an initiating event, whose frequency is also quantified by an uncertainty distribution. The term *Monte Carlo* is used to describe this analysis by repeated random sampling. Two kinds of Monte Carlo sampling are simple Monte Carlo sampling and Latin Hypercube sampling; they are described and compared in Sections 9.6 through 9.8.

Basic Uncertainty Output

The Monte Carlo procedure computes the probability distribution of a fault tree top event or accident sequence using the assigned probability distributions for each basic event contained in the minimal cut sets. By using the probability distributions for the basic events, the uncertainty in the system unavailability can be calculated.

The first step in the process of computing the uncertainty in the minimal cut set upper bound is to provide a measure of the uncertainty for each basic event contained in the minimal cut sets. SAPHIRE then computes the minimal cut set upper bound for a set of random samples from the uncertainty distributions of the basic events. After calculating the minimal cut set upper bound, SAPHIRE computes the first four moments of the distribution and the 5th, 50th, mean, and 95th percentile values.

The moments are calculated as a basis for comparison of the calculated distribution with other distributions (McGrath and Irving 1975). From the first four moments, the sample mean, sample variance, coefficient of skewness, and coefficient of kurtosis can be calculated. To establish some standard notation, the following symbols are used:

n	=	the number of samples calculated.
xi	=	*i*th data value for *i* = 1, 2, 3, ... n.

The sample mean, given as $\overline{x}$, can be defined as

$$\overline{x} = \sum_{1}^{n} \frac{x_i}{n} \qquad (54)$$

and the sample variance, given as

$$s^2 = \sum_{1}^{n} \frac{(x_i - \overline{x})^2}{n - 1} .$$

(55)

The *k-th* sample moment about the mean is next defined in general as

$$m_k = \sum_{1}^{n} \frac{(x_i - \overline{x})^k}{n - 1} \quad . \tag{56}$$

Thus, from the third moment, the coefficient of skewness, β11/2, is

$$\beta_1^{1/2} = \frac{m_3}{s^3} \tag{57}$$

and from the fourth moment, the coefficient of kurtosis, β2, is

$$\beta_2 = \frac{m_4}{s^4} \tag{58}$$

where *s* = the square root of *s2*.

The coefficient of skewness and the coefficient of kurtosis are generally used as measures for comparison with the normal distribution. If the skewness is close to zero while the kurtosis is approximately three, the normal distribution is a good approximation. A zero skewness value indicates a symmetric distribution; a negative skewness indicates a long left tail, while a positive value indicates a long right tail. If the kurtosis is greater than three, the distribution is more peaked than the normal distribution, and has more weight in the tails. However, if the value is less than three, the distribution is flatter than the normal, and has less weight in the tails.

Uncertainty Analysis Input Data

The parameters for the probability of a basic event, discussed in Section 7, are input in the Failure Data area of the SAPHIRE input screen. Now we move to the Uncertainty Data area using the arrow keys or the tab key. The fields in this area that can be accessed from this menu are the current case distribution type, a distribution parameter value, and a correlation class.

Currently, SAPHIRE supports lognormal, normal, beta, gamma, chi-squared, exponential, uniform, and histogram distributions for the Monte Carlo uncertainty analyses. The default distribution type is the lognormal.

Most distributions can be defined with two statistical parameters, although some take more. The first parameter is the mean failure probability and the second parameter is specific to the particular uncertainty distribution. The mean failure probability is calculated from the data input in the Failure Data area just discussed. For more clarity, SAPHIRE allows the user to input the parameters of the distribution directly. It will check them for consistency with the mean.

Correlation classes, as explained in Section 9.5, are used to identify basic events whose failure data are derived from the same data source. This information is used in the uncertainty analysis. Correlation classes consist of four upper-case values. A blank correlation class indicates that there are no data dependencies. When running the uncertainty analyses, the same sample value will be used for all basic events with the same correlation class.

> **NOTE**: The user must set up a correlation class numbering scheme for the basic events in the database. For example, correlation class 1 may be assigned to motor-driven pumps fail to start, correlation class 2 to motor-driven pumps fail to continue to run, correlation class 3 to check valves fail to close, and so on. Currently, this scheme is not saved within SAPHIRE but a feature to save it may be included in the future.

SAPHIRE provides more sophisticated ways of entering failure and uncertainty data that reduce the amount of data input required and ensure consistency among like basic events. These techniques are discussed in the *SAPHIRE Reference Manual* (Russell et al. 1992a).

Distributions

At the present time, as mentioned above, the following uncertainty distributions are supported: lognormal, normal, beta, gamma, chi-squared, exponential, uniform, and histogram. The histogram distribution requires detailed information to be fully specified. Each of the other distributions is described by its mean and typically one additional parameter. Table 2 summarizes this information for each of the supported distributions except for the histogram distribution, which is explained separately in Section 9.4. The distributions in Table 2 are described in Sections 9.3.1 through 9.3.7. More detail about these distributions can be found in Mood et al. (1974) and Hahn and Shapiro (1967).

Distribution	**Code**	**Parameter**
lognormal	L	95% error factor
normal	N	standard deviation
beta	B	*b* in beta(*a*, *b*)
Dirichlet	D	
gamma	G	*r* in gamma(*r*)
chi-squared	C	degrees of freedom
exponential	E	-
uniform	U	upper end point

maximum entropy	M	a = lower end range
		b = upper end range

One method for generating random numbers, called the *inverse c.d.f. method*, is used for several distributions below, and therefore is described here. Let X denote a random variable, let x denote a number, and let F denote the cumulative distribution function (c.d.f.) of X. It follows directly from the definition

$$F(x) = P(X \le x)$$

that $F(X)$ is a uniformly distributed random variable between 0 and 1. Therefore, generate U from a uniform distribution between 0 and 1, and solve $F(X) = U$ for $X = F^{-1}(U)$.

For example, if X is exponentially distributed with mean μ, the c.d.f. is

$$F(x) = 1 - e^{-x/\mu} \ .$$

Therefore, to generate an exponentially distributed random variable X, generate a uniformly distributed random variable U and let $X = F\text{-}1(U) = \mu\ln(1\text{-}U)$. Actually $\ln(U)$ can be used instead of $\ln(1\text{-}U)$, because if U is uniformly distributed between 0 and 1, then so is $1\text{-}U$.

The inverse c.d.f. method is only one of many methods of generating random numbers from a specified distribution. For some distributions it is natural and fast, and for other distributions a different method may be quicker. If the inverse c.d.f is hard to compute, for example if it must be found at any point by numerical iteration on the (non-inverse) c.d.f., then the inverse c.d.f. method is not a fast way to generate random numbers.

There is one application where the inverse c.d.f. method is very natural. This is in Latin Hypercube Sampling (LHS), where stratified portions of the distribution must be sampled. For example, if 20 points are to be sampled, one point must be below the 5th percentile, one must be between the 5th and the 10th percentiles, one between the 10th and 15th, and so forth. It is easy to sample in this way from a uniform distribution: For example, to sample a uniform (0, 1) distribution between its 10th and 15th percentiles, we must sample it and obtain a number between 0.10 and 0.15. Do this by letting U be uniform between 0 and 1. Then let Y equal $0.10 + 0.05U$, which is between 0.10 and 0.15. Then $X = F^{-1}(Y)$ is between the 10th and 15th percentiles of F, as required. For this reason, all Latin Hypercube samples are generated in SAPHIRE using the inverse c.d.f. method.

Lognormal Distribution

X has a lognormal distribution if $\ln X$ has a normal distribution. The parameters used in SAPHIRE to describe the lognormal distribution are the mean of the lognormal distribution and the upper 95% error factor. The mean value of the lognormal distribution, m, can be expressed as:

$$m = e^{\mu + \frac{\sigma^2}{2}} \qquad (9\text{-}1)$$

where μ is the mean and F is the standard deviation of the underlying normal distribution. Likewise, the 95% error factor (ef) for the lognormal distribution is given by

$$ef = e^{1.645\sigma} \qquad (9\text{-}2)$$

where 1.645 is the 95th percentile of the standard normal distribution. The density of the lognormal distribution is

$$f(x) = \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-[\ln(x)-\mu]^2/2\sigma^2}$$

for $x > 0$.

In SAPHIRE, a random variable X is sampled from the lognormal distribution as follows. Equations (9-1) and (9-2) are first solved for μ and σ. A random variable Y is generated from a normal distribution with mean μ and standard deviation σ, as explained in Section 9.3.2. Then X is defined as $X = \exp(Y)$. This is the procedure for simple Monte Carlo sampling and for Latin Hypercube sampling.

Normal Distribution

The additional parameter to describe the normal distribution in SAPHIRE is the standard deviation of the distribution, σ. The density function is given by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$$

where $-\infty<x<+\infty$.

SAPHIRE uses the Marsaglia-Bray algorithm, described on p. 203 of Kennedy and Gentle (1980), to generate a normal(0, 1) random variable Z. Then X, a normal random variable with mean μ and standard deviation σ, is defined as $X = \mu + \sigma Z$.

For LHS sampling from a normal distribution, the inverse c.d.f. method is used, with the inverse of the normal c.d.f. F-1(U) computed as follows. For $0.1 \le U \le 0.9$, F-1 is found by the algorithm of Beasley and Springer (1977). For $U < 0.1$ or $U > 0.9$, F-1 is approximated by Algorithm 5.10.1 of Thisted (1988), due to Wichura. The approximation is then refined by one application of Equation (5.9.2) of Thisted.

Beta Distribution

The parameters of the beta distribution are a and b. The probability density function is given by

$$f(x) = \left[\frac{1}{\mathrm{B}(a,b)}\right] x^{a-1}(1-x)^{b-1}$$

for $0 < x < 1$, where B(a,b) is the beta function. In SAPHIRE, the parameter b is used in addition to the mean to define the distribution. The parameter a is calculated from the mean value by the formula

$$a = \mu * b / (1- \mu)$$

where $\mu = a/(a+b)$ is the mean of the beta distribution. Note that the mean of the beta distribution is between 0 and 1.

SAPHIRE generates a beta random variable using the fact that if X is $\chi^2(2a)$ and Y is $\chi^2(2b)$ and X and Y are independent then $X/(X + Y)$ has a beta(a, b) distribution. See Section 24.2 of Johnson and Kotz (1970).

For LHS sampling, the inverse c.d.f. method is used, with the inverse of the c.d.f. computed by numerical iteration (with the method of false position) on the beta c.d.f. The beta c.d.f. is evaluated using the BETAI function of Press et al. (1986). Note, this way of generating the LHS sample is not fast, and simple Monte Carlo sampling with a larger sample may be more efficient than LHS sampling when many beta distributions must be sampled. Comparative tests have not been run.

Dirichlet Distribution

A k-variate random vector, ($X1$, $X2$, ... , Xk) is *Dirichlet* distributed with parameters $\alpha 1$, $\alpha 2$, ... αk and αk+1 if it has joint probability density function

$$f(x_1, x_2, \ldots, x_k) = \frac{\Gamma(\alpha_1 + \alpha_2 + \ldots + \alpha_{k+1})}{\Gamma(\alpha_1)\Gamma(\alpha_2)\ldots\Gamma(\alpha_{k+1})} x_1^{\alpha_1 - 1} x_2^{\alpha_2 - 1} \ldots x_k^{\alpha_k - 1} (1 - x_1 - x_2 - \ldots - x_k)^{\alpha_{k+1} - 1}$$

for $0 < xi$ and $x1 + x2 + \cdots + xk < 1$.

The Dirichlet distribution is a generalization of the Beta distribution with parameters a and b if we let $k = 1$, $a = \alpha 1$ and $b = \alpha 2$. The mean of the ith Dirichlet variable, Xi, is

$$\mu i = \alpha i/D$$

where the denominator D is defined as

$$D = \alpha 1 + \alpha 2 + \cdots + \alpha k+1.$$

Thus, an alternative parameterization is in terms of the means $\mu 1$, $\mu 2$, ... , μk and either D or $\beta \equiv \alpha k$+1. These are equivalent parameterizations in the sense that each can be solved uniquely in terms of the other. For example, if the μis and β are given, then

$$D = \frac{\beta}{1 - \sum_{i=1}^{k} \mu_i}$$

and

$\alpha_i = \mu_i D$ for $i = 1, 2, \ldots, k$.

The marginal distributions of the individual Dirichlet variables are Beta distributions. In particular, the ith variable, X_i, has the Beta distribution with parameters $a = \alpha_i$ and $b = D - \alpha_i$.

A set of k-variate Dirichlet variables are dependent random variables, but it is possible to transform from a set of k independent Beta distributed random variables into dependent Dirichlet random variables. Specifically, if $Y_1, \ldots, Y_k$ are independent random variables, such that the ith variable, Y_i, has a Beta distibution with parameters $a = \alpha_i$ and $b = D - \alpha_1 - \cdots - \alpha_{i-1}$, then the vector of k random variables $(X_1, X_2, \ldots, X_k)$ defined by $X_1 = Y_1$ and

$$X_i = (1 - X_1 - \cdots - X_{i-1})Y_i \quad \text{for } 2 \le i \le k,$$

are distributed as k-variate Dirichlet with parameters $\alpha_1, \alpha_2, \ldots, \alpha_k$ and β.

It is also true that a subset of Dirichlet variables is also Dirichlet distributed. For example, if $1 \le i < j \le k$, then the pair (X_i, X_j) has the 2-variate Dirichlet distribution with parameters α_i, α_j and $D - \alpha_i - \alpha_j$.

More generally, suppose we define the set $E = \{1, 2, \ldots, k, k + 1\}$, and denote a subset by $F = \{i_1, i_2, \ldots, i_m\} \subset E$. Then the vector $(X_{i_1}, X_{i_2}, \ldots, X_{i_m})$ has the m-variate Dirichlet distribution with parameters $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_m}$ and $D - \alpha_{i_1} - \alpha_{i_2} - \cdots - \alpha_{i_m}$.

EXAMPLES

For example, suppose a branch in an event tree has three mutually exclusive failure events A_1, A_2, A_3 and the complement (success) event B. Suppose the uncertainty in the probabilities of these events is modeled using a 3-variate Dirichlet distribution. In other words, P(A_1), P(A_2) and P(A_3) are modeled as Dirichlet with nominal values μ_1, μ_2 and μ_3. The remaining parameter β is the nominal value of P(B). The alternate parameterization in terms of D is useful, because it is involved in the variances of the individual Dirichlet variables. Specifically, the ith variable has variance $\mu_i(1 - \mu_i)/(D + 1)$, so a smaller value of D correponds to a large variance, which in turn means greater uncertainty about P(A_i). Similarly, a large value of D corresponds to a small variance or less uncertainty. Variables with this distribution can be generated by first generating <u>independent</u> Beta variables, Y_1 with parameters $a = \alpha_1$ and $b = D - \alpha_1$, Y_2 with parameters α_2 and $b = D - \alpha_1 - \alpha_2$, and Y_3 with parameters $\alpha 3$ and $D - \alpha 1 - \alpha 2 - \alpha 3 = \beta$. Then, the required Dirichlet variables would be $X1 = Y1$, $X2 = (1 - X1)Y2$ and $X3 = (1 - X1 - X2)Y3$.

If only one branch (say the first) is of interest, then the uncertainty can be given in terms of a Beta variable with mean $\mu 1 = \alpha 1/D$ and $b = \alpha 2 + \alpha 3 + \beta = D - \alpha 1$. Suppose it is required to consider jointly two of the events (say $A1$ and $A2$). The "joint" uncertainty is modeled by a 2-variate Dirichlet distribution with parameters $\alpha 1$, $\alpha 2$ and $\beta^* = D - \alpha 1 - \alpha 2 = \alpha 3 + \beta$.

REFERENCE

Wilks, S. S. (1962), *Mathematical Statistics*, John Wiley and Sons: New York.

Gamma Distribution

The parameters of the gamma distribution are λ and r. The probability density function is given by

$$f(x) \;=\; \frac{\lambda^r}{\Gamma(r)} x^{r-1} e^{-\lambda x}$$

for $x > 0$, where ’(r) is the gamma function. In SAPHIRE, the value in the uncertainty distribution is r. The parameter $\lambda$is calculated from the mean value by the formula $\lambda = r/\mu$, since the mean is $\mu = r/\lambda$.

SAPHIRE generates a gamma random variable in two stages. First it generates a random variable Y from a gamma distribution with the desired r and with $\lambda = 1$. The algorithm used depends on the value of r: if $r < 1$, SAPHIRE uses Algorithm GS of Ahrens and Dieter (1974); if $r > 1$, SAPHIRE uses Algorithm GA of Cheng (1977); finally, if $r = 1$, SAPHIRE uses the inverse c.d.f. method explained at the beginning of Section 9.3. Once Y has been generated, the gamma random variable with parameter r and with the desired mean μ is defined as $X = Y/\lambda$, with $\lambda = r/\mu$.

For LHS sampling, SAPHIRE uses the fact that the gamma and the chi-squared distributions are different parameterizations of the same distribution. SAPHIRE uses the inverse c.d.f. method described in Section 9.3.5 to generate LHS samples from a gamma distribution.

Chi-Squared Distribution

The chi-squared distribution is directly related to the gamma distribution, as follows. Let X have a gamma(λ, r) distribution. Then $2\lambda X$ has a chi-squared distribution with $2r$ degrees of freedom, denoted $\chi 2(2r)$. For this reason, the chi-squared distribution is an option in

SAPHIRE only as a convenience to the user. Anything that requires a chi-squared distribution can be accomplished using a gamma distribution.

The mean of a $\chi2(k)$ distribution equals k and the variance equals $2k$, for degrees of freedom $k > 0$. Note that the mean of a chi-squared distribution determines the variance. This is not flexible enough for most uncertainty analyses. Therefore, when SAPHIRE is asked for a chi-squared random variable with k degrees of freedom and mean :, it generates a *multiple* of a chi-squared random variable, $Y = aX$, where X is $\chi2(k)$ and $a = \mu /k$. This results in a random variable with mean μ and variance $2\mu\ 2/k$. Exactly the same distribution would be obtained by specifying a gamma distribution with mean μ and $r = k/2$.

For simple Monte Carlo sampling, SAPHIRE generates a multiple of a chi-squared random variable with mean μ and k degrees of freedom by generating a gamma random variable with mean μ and with $r = k/2$.

For LHS sampling , SAPHIRE considers several special cases, and uses the inverse c.d.f. method in every case. Let k be the degrees of freedom. When $k = 1$, the random variable is the square of a normal random variable, and SAPHIRE finds the inverse c.d.f. based on the inverse of a normal c.d.f. When $k = 2$, the distribution is exponential, and SAPHIRE finds the inverse c.d.f. explicitly. For all other values of k, SAPHIRE begins with the Wilson-Hilferty approximation of the chi-squared inverse c.d.f. (Section 5.10.2. of Thisted 1988), or 0 if the Wilson-Hilferty approximation is negative. It then refines this approximation by numerical iteration (the method of false position), obtaining values for which the chi-squared c.d.f. is ever closer to the specified value of the c.d.f. The chi-squared c.d.f. is computed as the equivalent gamma c.d.f., calculated using the algorithm GAMMP given by Section 6.2 of Press et al. (1986). When the Wilson-Hilferty approximation is good, typically in the right tail of the distribution with k not very small, very little time is required for the refinement.

Exponential Distribution

The exponential distribution is commonly used for modeling a time to failure, but it is not very useful for modeling uncertainties, and may some day be dropped as an option in this part of SAPHIRE. One reason for its use in modeling failures and its disuse in modeling uncertainties is that it has only one parameter. Therefore the mean determines the variance. The exponential density is

$$f(x) = \lambda e^{-\lambda x}$$

where the parameter λ and the mean μ are related by $\mu = 1/\lambda$. Note that the exponential density is a special case of the gamma density, with the gamma parameter $r = 1$. Alternatively, if Y is $\chi2(2)$, then $X = Y/(2\lambda)$ has a gamma distribution with $r = 1$ and mean $\mu = 1/\lambda$, i.e. an exponential(λ) distribution. Therefore, anything that can be simulated with an exponential distribution can also be simulated with a gamma or chi-squared distribution.

An exponential(λ) random variable is generated by the inverse c.d.f. method, as explained at the beginning of Section 9.3. This method is recommended in Section 6.5.2 of Kennedy and Gentle (1980) for the gamma distribution with $r = 1$.

Uniform Distribution

The mean of this distribution is $M = (a+b)/2$. The value in the uncertainty distribution in SAPHIRE is b, the right (upper) endpoint of the distribution. The value for a is calculated by the equation $a = 2*M - b$. The density function for this distribution is

$$f(x) = \frac{1}{b - a}$$

for $a \leq x \leq b$.

SAPHIRE generates a uniformly distributed random number using the prime modulus multiplicative linear congruential generator advocated by Park and Miller (1988). The modulus m is 231-1 = 2,147,483,647 and the multiplier is 16807. This generates a sequence of m - 1 distinct integers before repeating, in an order that appears random. To obtain real numbers between 0 and 1, the integer obtained in this way is divided by m.

Having generated a random variable Y uniform between 0 and 1, SAPHIRE obtains a random number uniform between a and b as $X = a + (b-a)Y$. This is used for both simple Monte Carlo sampling and for LHS sampling.

Maximum Entropy Distribution

The maximum entropy distribution is specified by its mean value, μ, and by a, the lower end of the range, and b, the upper end of the range. Section B1 of Appendix B shows that if μ is not at the midpoint of the range, the density has the form

$$f(x) = \beta e^{\beta x} / (e^{\beta b} - e^{\beta a}) \tag{9-3}$$

for $a \leq x \leq b$, with β a non-zero parameter. If β is negative, the distribution is a truncated exponential distribution with parameter $-\beta$. If β is positive, the density is of exponential form, an increasing function of x. If instead μ is the midpoint of the range, $\mu = (a + b)/2$, then the maximum entropy density is not of the form (9-3), but instead is flat, a uniform density. This is the limiting value for (9-3) as $\beta \rightarrow 0$.

The density (9-3) corresponds to a c.d.f. of the form

$$F(x) = (e^{\beta x} - e^{\beta a}) / (e^{\beta b} - e^{\beta a}) . \tag{9-4}$$

The parameter β and the user-input mean μ are related by

$$\mu = (be^{\beta b} - ae^{\beta a}) / (e^{\beta b} - e^{\beta a}) - 1/\beta , \text{ for } \beta \neq 0 . \tag{9-5}$$

Section B1 of Appendix B shows that μ is a monotonic function of β. As $\beta \to -\infty$, μ approaches its minimum possible value, *a*, and as $\beta \to \infty$, μ approaches its maximum possible value, *b*. This monotonicity is the basis for the algorithm given in Section B2 of Appendix B for finding β from the user inputs of μ, *a*, and *b*.

When the user specifies a maximum entropy distribution with mean μ and range [*a*, *b*], SAPHIRE first uses a numerical search to obtain the value of β satisfying Equation (9-5). It then inverts Equation (9-4) explicitly and uses the inverse c.d.f. method, by generating a random number *U* from a uniform distribution between 0 and 1, and calculating

$$X = \{ \ln[(e^{\beta b} - e^{\beta a})U + e^{\beta a}] \} / \beta ,$$

a random number from the maximum entropy distribution. This method is used both for simple Monte Carlo sampling and for LHS sampling.

Histograms

SAPHIRE allows for either a discrete or a continuous distribution under this option. The modeled quantity is a probability λt or $\lambda\tau$, or a frequency $f\lambda t$ or $f\lambda\tau$. When the PERCENT option is selected, the distribution is discrete on up to 20 values; the percents, giving the degree of belief for each value, must sum to 100. If the RANGE or AREA option is selected, the density is a step function covering up to 20 adjacent intervals. The function is constant within each interval, and the area under the entire function must equal 1.0.

Correlation Classes

The practice of using the same uncertainty distribution for a group of similar components has been common since the Reactor Safety Study (NRC 1975). The PRA Procedures Guide (Hickman 1983) recommends this practice as well. Philosophical arguments have been given to support this practice or used to give it credence. Apostolakis and Kaplan (1981) discuss this issue from a Bayesian perspective, and they call it a "lack of knowledge" dependency. However, this dependency is broader than just a lack of knowledge. It is present whenever the same data set is used for several components. It is not a Bayesian or classical statistical phenomenon, but it is induced because of the way the data are used.

For example, suppose that a plant has two motor-driven AFW pumps. These pumps are virtually identical, and therefore are modeled as having the same unavailability, *q*. The uncertainty distribution for *q* is taken from some database, and describes our best belief about the true value of *q*. Because the two components have uncertainty distributions taken from the same source, if our estimate of *q* is too high (say) for one pump, it will be also be too high for the other pump, by the same amount. Similarly, if our estimate is too low for one, it will be too low for the other by the same amount. The uncertainty distributions for the two unavailabilities are perfectly correlated.

This correlation of the uncertainties must be distinguished from the independence of the basic events. The two basic events (failures of the pumps to be available) are independent; that is, the probability that one pump is unavailable is some number *q*, unaffected by whether the other pump is available or not. However, our uncertainty about the value of *q* is totally correlated for the two basic events.

The user tells SAPHIRE of this uncertainty correlation by putting the two basic events in a single *correlation class*. When *q* is sampled from its uncertainty distribution, that one value of *q* is assigned to all the basic events in the correlation class. After the probability of the top event has been calculated, on the next Monte Carlo pass a new (presumably different) value of *q* is drawn from the uncertainty distribution, and is assigned to all the basic events in the class.

Let us now examine the effect of total correlation in accident sequence analysis. Consider a simple example involving a cut set with two components. Let *q1* and *q2* denote the unavailability of the two components in the cut set. If the components are independent, then

$$Q = q_1 q_2 \tag{9-6}$$

is the cut set unavailability.

As we begin the analysis, we can make one of two assumptions. First, we can assume that the unavailability of each component is estimated from independent data sources. For example, if the first basic event is failure of a pump and the second basic event is failure of a valve, the probabilities of these basic events will be estimated from independent sources, and therefore the two probabilities have independent uncertainty distributions. The expected value and variance of *Q* are given by

$$E(Q) = E(q_1)E(q_2) \tag{9-7}$$

and

$$var(Q) = E(q_1^2)E(q_2^2) - \left[E(q_1)E(q_2)\right]^2 \quad . \tag{9-8}$$

These equations follow from the independence of the uncertainty distributions.

If instead, the components are identical, then *q1 = q2 = q*, and Equations (9-7) and (9-8) reduce to

$$E(Q) = E(q_1)E(q_2) = \left[E(q)\right]^2 \tag{9-9}$$

and

$$var(Q) = \left[E(q^2)\right]^2 - \left[E(q)\right]^4 \quad . \tag{9-10}$$

However, when the components are identical, Equations (9-9) and (9-10) are probably not correct. The same source would presumably be used to obtain the uncertainty distribution for both unavailabilities. Therefore, any value *q* that is used for one basic event should also be used for the others. Equation (9-6) reduces to

$$Q = q^2 \quad ,$$

so we have

$$E(Q) = E(q^2) \tag{9-11}$$

and

$$var(Q) = E(q^4) - \left[E(q^2)\right]^2 . \tag{9-12}$$

A standard identity from statistics says that

$$E(q^2) = [E(q)]^2 + var(q) > [E(q)]^2 .$$

Therefore, Equation (9-11), the correct one, is larger than Equation (9-9), the incorrect one. This is why the point estimate and the mean of the uncertainty distribution are not equal in PRAs. The point estimate for the example cut set is the product of the basic event means, given by Equation (9-9), whereas the mean of the cut set uncertainty distribution is given by the larger value in Equation (9-11). Similarly, the variance should be calculated from Equation (9-12), not Equation (9-10). In typical cases, including any case in which *q* is lognormally distributed, Equation (9-12) gives a larger value than Equation (9-10). The effects are most pronounced when the distributions are highly skewed.

Ericson et al. (1990, page 12-8) suggests the following steps for grouping basic events into correlation classes:

- Group all basic events by component type (e.g., MOV, AOV, MDP),

- Within each component group, organize events into subgroups by failure mode (e.g., fail-to-start, fail-to-run),

- For time related basic events, group all events from each component failure mode group into sets according to the time parameter value used to quantify the event probability (e.g., 6 hours, 720 hours), and

- For demand related failures, no further grouping is necessary beyond the component failure model level.

If different estimates are developed for components within the same component group (e.g., Service Water Motor-Driven Pump, Residual Heat Removal Motor-Driven Pump), then these should be treated as separate component groups.

Sampling Techniques

Overview of Simple Monte Carlo Sampling

The Monte Carlo approach is the most fundamental approach to uncertainty analysis. Simple Monte Carlo simulation consists of making repeated quantifications of the top event value using values selected at random from the uncertainty distributions of the basic events. For each iteration of the Monte Carlo run, each basic event uncertainty distribution is sampled using a random number generator to select the failure probability of the basic event. The top event probability or accident sequence frequency is calculated. When this procedure has been repeated a predetermined number of times, the top event or accident sequence results are sorted to obtain empirical estimates of the desired top event attributes such as the mean, median, 5th percentile, and 95th percentile. A plot of the empirical uncertainty distribution is often obtained. 20 contains an example of an uncertainty distribution for an accident sequence. For more information about the Monte Carlo technique the reader is referred to Hahn and Shapiro (1967).

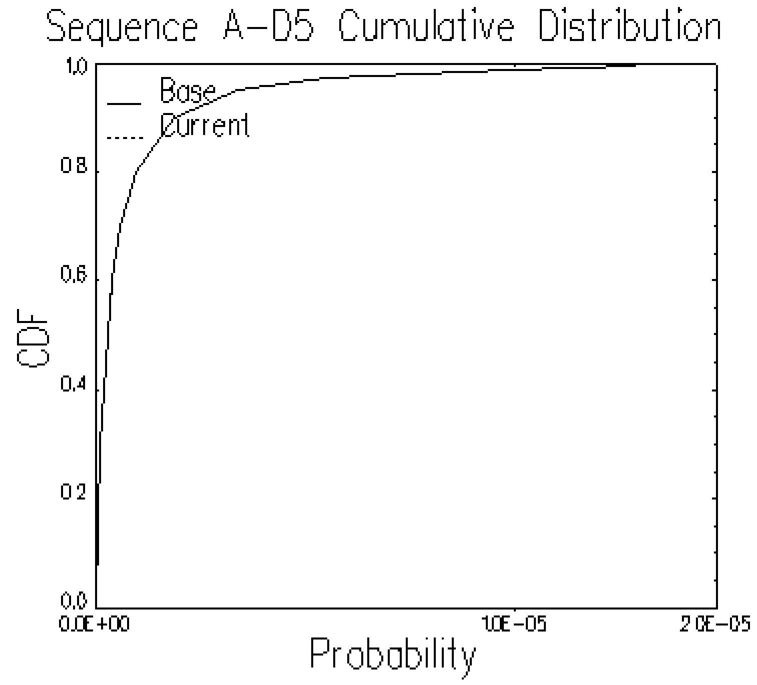


Figure 20. Uncertainty distribution for an accident sequence.

To illustrate the Monte Carlo technique, consider a system with two components in series. Let *A* denote failure of the first component and *B* failure of the second. The cut sets for the system are *A* and *B*, so the equation for the top event (system) is

$$S = A + B$$

Let *A* and *B* have mean failure probabilities of 0.001 and 0.005, respectively. Also assume that the uncertainty distribution for *A* is uniform from 0 to 0.002 and the distribution for *B* is normal with standard deviation of 0.001. These distributions are shown in 21 and 21.

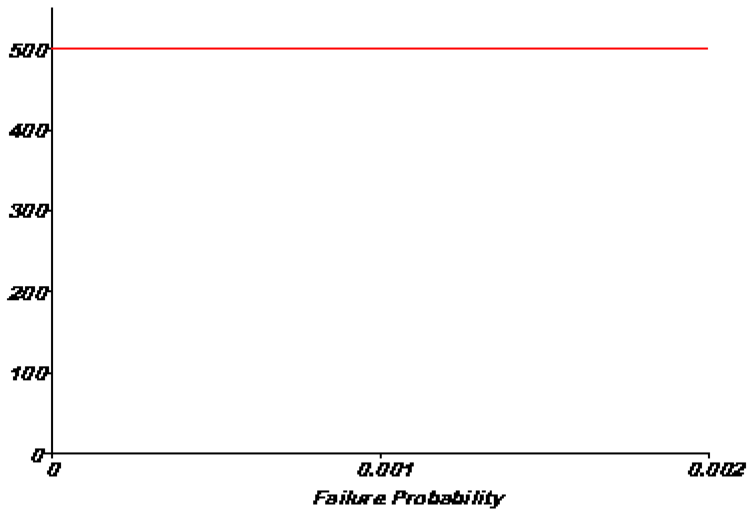


Figure 21. Uncertainty distribution for Component A

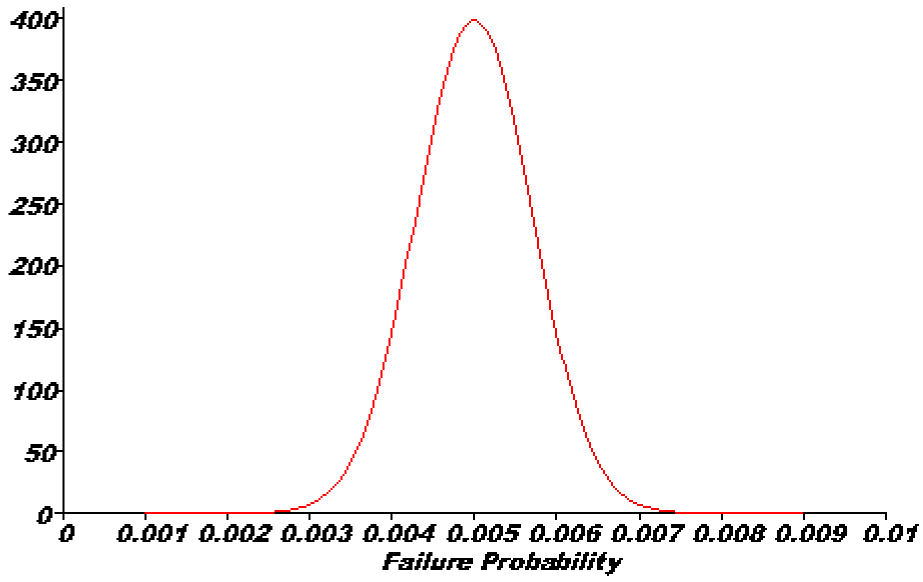


Figure 22. Uncertainty distribution for Component B

The point estimate for *S* is 0.006. Table 3 contains a random sample of size 10 for this example. Column 1 contains the sample for component A which has a uniform uncertainty distribution. Column 2 contains the sample for failure of component *B*, and column 3 contains the sum of columns 1 and 2 which is the minimum cut set upper bound for the probability of failure of the system. The bottom row is the average of the columns.

A	**B**	**A+B**
0.00042	0.00500	0.00542
0.00086	0.00661	0.00747

0.00149	0.00570	0.00719
0.00109	0.00605	0.00714
0.00066	0.00420	0.00487
0.00024	0.00609	0.00633
0.00066	0.00396	0.00462
0.00075	0.00293	0.00368
0.00037	0.00500	0.00537
0.00127	0.00597	0.00724
0.00078	0.00515	0.00593

Overview of Latin Hypercube Sampling

Latin Hypercube Sampling (LHS) selects *n* different values from each of the *k* variables *X1*,...,*Xk* in the following manner. The range of each variable is divided into *n* nonoverlapping intervals on the basis of equal probabilities for the intervals. The *n* values thus obtained for *X1* are paired in a random manner with the *n* values of *X2*. These *n* pairs are combined in a random manner with the *n* values of *X3* to form *n* triplets, and so on, until *n k*-tuplets are formed. This is the Latin Hypercube sample. It is convenient to think of the LHS, or a random sample of size *n*, as forming an *n x k* matrix of inputs where the *i*th row contains specific values for each of the *k* input variables to be used on the *i*th evaluation of the cut sets.

To help clarify how intervals are determined in the LHS, consider the simple example used in the previous section. We want to generate an LHS sample of size 5. The first step is to divide the uncertainty distributions of *A* and *B* into 5 equal probability areas each containing an area of 0.2. For *A* this is easy since it has a uniform uncertainty distribution. The points separating the cells are 0.0004, 0.0008, 0.0012, and 0.0016. The areas are shown in 23. The uncertainty distribution for *B* is a normal distribution; it is harder to find the points that divide the areas into equal probability areas. Probability tables or a calculator with an inverse normal calculation routine is needed. The four points which define the 5 equal probability areas are 4.158E-3, 4.747E-3, 5.253E-3, and 5.842E-3. These are shown in 23.

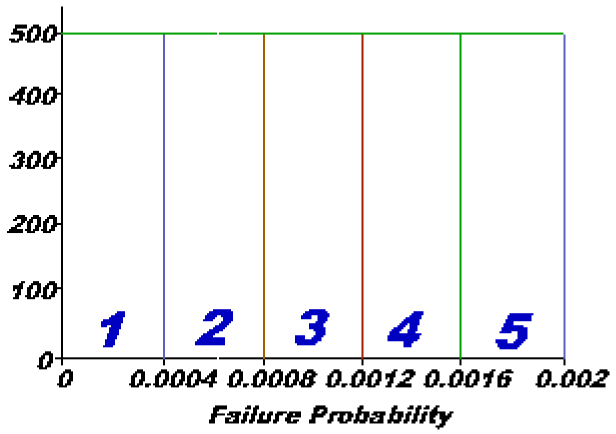


Figure 23. Latin hypercube sample for Component A

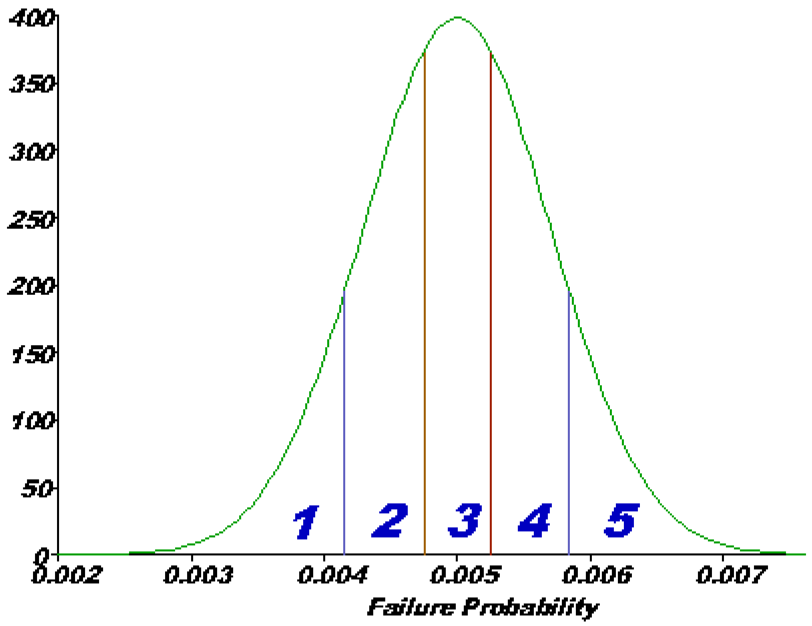


Figure 24. Latin hypercube sample for Component B

The next step is to generate a random permutation of the integers (1, 2, 3, 4, 5). In this example we get (2, 3, 4, 1, 5). Now we choose a random offset for each basic event, 2 for A and 3 for B. We start the basic permutation for A at the second element, yielding (3, 4, 1, 5, 2), and we start the basic permutation for B at the third element, yielding (4, 1, 5, 2, 3). Combining these yields:

Computer Run	Interval for A	Interval for B
1	3	4
2	4	1
3	1	5
4	5	2
5	2	3

These five cells are shown in Figure 25. The next step is to obtain random values for *A* and *B* for each of the intervals. The first value for *A* lies in interval 3; thus, the value must be between 0.0008 and 0.0012. *A* is generated as described in Section 9.3.7: A random number *U* is generated from a uniform distribution between 0 and 1. Then *A* is defined as 0.0008 + 0.0004*U*. The corresponding value for *B* lies in interval 4; thus the value for *B* must lie between the 60th and 80th percentiles of the normal distribution. This is generated as described in Section 9.3.2: A new random number *U* is generated from a uniform distribution between 0 and 1, and $V = 0.6 + 0.2U$ is therefore uniform between 0.6 and 0.8. Let *F* denote the standard normal c.d.f. Then $Y = F$-1(V) is sampled from between the 60th and 80th percentiles of the standard normal c.d.f. Finally $B = 0.005 + 0.001Y$ is sampled from between the 60th and 80th percentiles of a normal distribution with mean 0.005 and standard deviation 0.001. The following table summarized the random numbers in this case.

Computer Run	Value for A	Value for B	Value for A+B
1	9.454E-4	5.398E-3	6.343E-3
2	1.512E-4	3.862E-3	5.374E-3
3	6.102E-5	6.684E-3	6.745E-3
4	1.827E-3	4.504E-3	6.331E-3
5	7.068E-4	4.898E-3	5.605E-3
Mean	1.010E-3	5.069E-3	6.080E-3

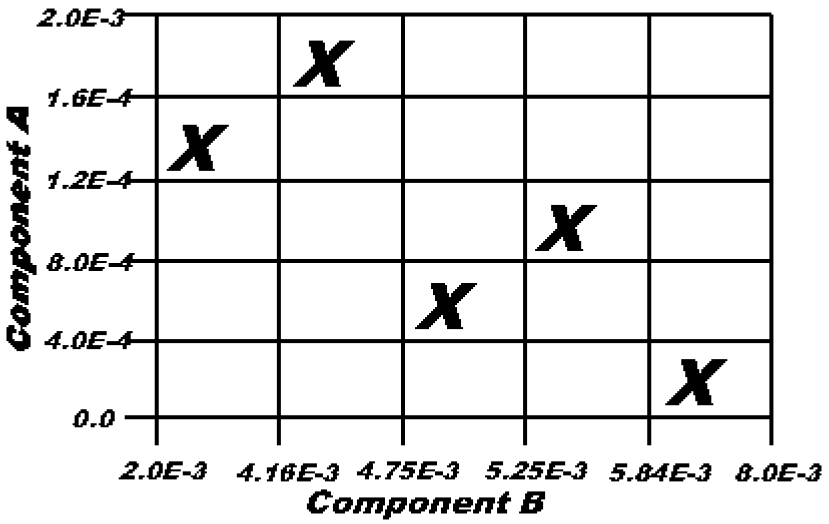


Figure 25. Cells sampled in LHS example.

This way of generating a permutation for each basic event, using a random offset of a single permutation, is less general than choosing a random permutation for each basic event. It is used in SAPHIRE to save storage and, secondarily, to save execution time.

Comparison of Simple Monte Carlo and Latin Hypercube Sampling

The following information is a comparison of Simple Monte Carlo simulation and Latin Hypercube Sampling (LHS). The table contains output from SAPHIRE for the sample problem in the previous section and also the exact values, which should be obtained if the sample sizes were infinite. 26 contains a plot of the cumulative distribution function for each sample. The results are very similar for these two methods. Notice the size of the samples for each. The LHS method requires only a quarter of the sample size of ordinary Monte Carlo, for similar accuracy. This must be balanced against the fact that for some distributions it takes longer to generate a random number for an LHS sample than for a simple Monte Carlo sample. Nevertheless, LHS sampling can often substantially reduce the time required for an analysis, while obtaining similar accuracy.

Table 4. Comparison of Monte Carlo and LHS for sample problem

	Monte Carlo	**LHS**	**Theoretical Exact**
Random Seed	51530	27290	-
Sample Size	200	50	-
Point estimate	5.995E-003	5.995E-003	6.000E-003
Mean Value	6.008E-003	5.994E-003	6.000E-003
5th Percentile Value	3.890E-003	3.876E-003	4.101E-003
Median Value	6.103E-003	6.320E-003	6.000E-003
95th Percentile Value	7.783E-003	7.816E-003	7.899E-003
Minimum Sample Value	2.798E-003	2.789E-003	-
Maximum Sample Value	8.944E-003	8.605E-003	-
Standard Deviation	1.163E-003	1.245E-003	1.155E-003
Skewness	-1.973E-001	-3.071E-001	0.000
Kurtosis	2.860E+000	2.747E+000	3.094
Elapsed Time	00:00:02.530	00:00:00.650	

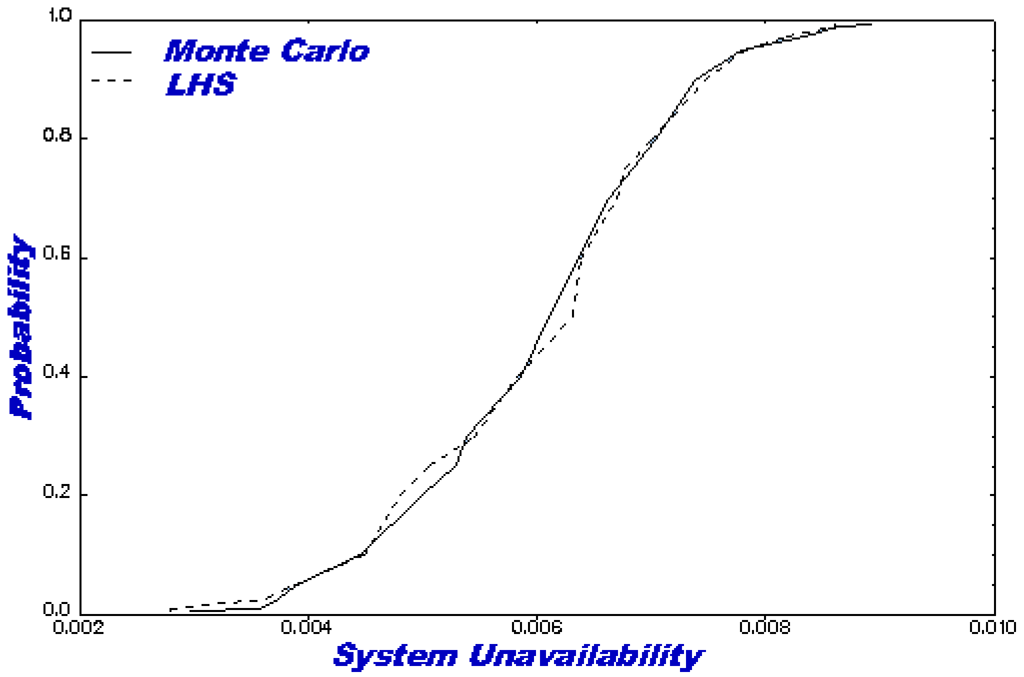


Figure 26. Cumulative distribution plots for example using Monte Carlo and LHS.

SEISMIC EVENTS

Fragility and Component Failure Probabilities

The severity of an earthquake is measured by its peak ground acceleration, denoted *g*. The *fragility distribution* of a component is the probability that the component fails, expressed as a function of *g*. The *fragility curve* is a graph of this function, plotting the probability of failure against the peak ground acceleration. For simplicity, the fragility is assumed to be based on a lognormal distribution. The approach is based on Kennedy et al. (1980).

For a particular component, let Af denote the "failure acceleration," the peak ground acceleration that is just sufficient to cause a component to fail. It is random, because sometimes a component will fail when subjected to a certain acceleration and sometimes an apparently identical component will survive when subjected to the same acceleration. The randomness is a really a property of the components. Af is random in the frequentist sense; that is, it is the random outcome of a hypothetical repeatable experiment in which we are assumed able to measure the acceleration that is just enough to cause the particular component to fail. This distribution is another way of expressing the fragility distribution, with large Af corresponding to small fragility.

Assume for the moment that there is no uncertainty, only randomness as described above. Uncertainty will be modeled later. Model the failure acceleration as a lognormal random variable:

$$A_f \;=\; \alpha\,\varepsilon_R \quad . \tag{10-1}$$

Here ϵR is a lognormal random variable such that ln(ϵR)/βR is normal(0,1), and α is the median failure acceleration. For now, α is treated as being perfectly known. The quantity βR is a parameter of the model. The subscript *R* stands for "random," and is a reminder that this probability distribution refers to random differences between nominally like components, not to subjective uncertainty. Another way to write (10-1) and the explanatory text is

$$\ln(A_f \;/\alpha)/\beta_R \quad normal(0,1) \quad . \tag{10-2}$$

Consider some specific acceleration *g*. The probability of a component's failure, given that the component is subjected to this *g*, is:

$$\begin{aligned} P[\, g \ge A\text{f}\,] &= P[A\text{f} \le g\,] \\ &= P[\,\ln(A\text{f}/\alpha)/\beta\text{R} \le \ln(g/\alpha)/\beta\text{R}\,] \\ &= \Phi[\,\ln(g/\alpha)/\beta\text{R}] \end{aligned} \tag{10-3}$$

where Φ is the standard normal cumulative distribution function, and the last step follows from Equation (10-2). Equation (10-3) gives the fragility curve for the component, relating the assumed peak ground acceleration *g* to the probability that the component fails.

For a PRA without an uncertainty analysis, Equation (10-3) is used for the basic event failure probability, conditional on the acceleration *g*. The frequency of *g* is treated like the frequency of an initiating event.

Now consider uncertainty. Let *a* be the best estimate of the uncertain α. This is the quantity entered by the user in the field for median ‚Failure Acceler.' Model the uncertainty of α in the usual Bayesian way by treating α as a random variable with median *a*:

$$\alpha = a\,\varepsilon_U$$

where ϵU is a lognormal random variable such that ln(ϵU)/βU is normal(0,1). This can be rewritten as

$$\ln(\alpha / a) / \beta_U \quad normal(0,1) \ .$$

Note, the meaning of "best estimate" is different from the use in the non-seismic portions of SAPHIRE. Here, the best estimate is the median of the uncertainty distribution, whereas in the non-seismic analysis the best estimate is the mean of the uncertainty distribution. There are two reasons for this. (1) It follows Kennedy et al. (1980), and so is consistent with customary use by seismic analysts. (2) It is more conservative than using the mean. The mean is larger than the median, and corresponds to a larger failure acceleration, hence a smaller fragility. Use of the mean failure acceleration as a nominal value would result in a larger nominal failure acceleration, and a smaller nominal fragility. This would lead to more severe truncation of cut sets, and also a smaller nominal top event probability. This is the reverse of the usual, non-seismic, case. There, the failure probability, not a failure acceleration, has an uncertainty distribution, and use of the mean instead of the median results in a larger nominal failure probability, less severe truncation of cut sets, and a larger nominal top event probability.

In summary, the user inputs *a*, βR, and βU, where *a* estimates α. Here α is the median failure acceleration (the median of the distribution whose dispersion is determined by βR), and *a* is the median of the distribution quantifying uncertainty about α, the distribution whose dispersion is determined by βU. The uses by SAPHIRE of the input quantities are discussed next.

For many purposes, the nominal basic event probabilities are used. These include truncating cut sets, calculating the nominal top event probability, and calculating basic event importances. Conditional on a peak ground acceleration *g*, the nominal probability of any basic event is given by Equation (10-3) with *a* substituted for α.

When performing an uncertainty analysis in a PRA, it is customary to quantify the uncertainties in the failure probabilities by Monte Carlo simulation. This approach is followed in SAPHIRE. First, α is generated by Monte Carlo sampling, by letting *z* be generated by a standard normal random number generator, and letting

$$\alpha \ = \ a exp(\beta_U z) \ .$$

For a given g, the failure probability for a component is given by substituting this value of α into Equation (10-3).

A quantity of some interest is denoted HCLPF (pronounced hick-clip or hick-cliff), the maximum acceleration that corresponds to High Confidence of Low Probability of Failure. The quantity is calculated as

$$\text{HCLPF} = a\exp[-1.645(\beta R + \beta U)] \ ,$$

which is derived as follows. HCLPF requires both high confidence and low probability. The words “high” and “low” are taken to mean 0.95 and 0.05, respectively. Consider first the low-probability portion: In terms of the true, but unknown, median failure acceleration, α, the probability of a component's failure is given by Equation (10-3). The failure probability is low, that is, $\leq$ 0.05, if $\ln(g/\alpha)/\beta R$ at most the 5th percentile of the standard normal distribution. This says that

$$\ln(g/\alpha)/\beta R \ \leq \ -1.645 \ ,$$

from which we obtain

$$g \ \leq \alpha\exp(-1.645\beta R) \ . \qquad (10\text{-}5)$$

That is the low-probability portion of the definition. For the high-confidence portion, recall that α is not known, but has an uncertainty quantified by Equation (10-4). Therefore, with high (95%) confidence we have

$$\ln(\alpha/a)/\beta U \ \geq \ -1.645$$

or, equivalently,

$$a\exp(-1.645\beta U) \ \leq \ \alpha \ .$$

Substitute this bound on α into Equation (10-5). It follows that we have high confidence that Equation (10-5) is true if

$$g \ \leq a\exp(-1.645\beta U)\exp(-1.645\beta R) \ \ / \ a\exp[-1.645(\beta R + \beta U)] \ .$$

The largest such g is the expression on the right, the HCLPF value.

Frequencies of Seismic Events

For cut set truncation, the user enters a single peak ground acceleration, g. Equation (10-3) is used with this g and the best estimate of α to determine the nominal basic event probabilities. Cut sets are truncated based on these probabilities. It is a mistake to enter a small value of g, because too many cut sets will be truncated.

For all other portions of the analysis, the user may enter up to 20 peak ground accelerations gi that could be produced by an earthquake, and up to 20 corresponding frequencies, with units 1/yr. This assumes that the user knows the frequencies. To account for uncertainty in the frequencies in a simple way, three sets of accelerations and frequencies can be entered, low, medium, and high. The analysis can be based on the medium (best estimate) set, or on the low set (mild earthquake assumptions) or the high set (severe earthquake assumptions).

Recall that non-seismic analyses have initiating events, with event frequencies. Examples are loss of off-site power and a large LOCA. A seismic analysis differs, because each postulated gi is an initiator with its corresponding frequency. The events such as loss of off-site power and large LOCA are consequences of the earthquake. We will call them "induced initiators," but mathematically, they are events just like basic events. They have probabilities, not frequencies, and these probabilities are conditional on the value of gi. Similarly, the basic events have probabilities that depend on gi, found using Equation (10-3) above. In an accident sequence, the frequency of gi is multiplied by the probability of the induced initiator and by the probability of failure of each other component or system in the sequence, to yield the frequency of the sequence, with units 1/yr. The frequency of a plant state is the sum of the frequencies of the accident sequences (with various levels of gi) corresponding to that plant state, plus the frequency of the plant state found by a non-seismic analysis assuming no earthquake.

After setting up the initiators and the events as described above, SAPHIRE performs its seismic calculations in the same way that it performs calculations for non-seismic sequences.

SAPHIRE Seismic Implementation

The following discusses the features and use of the seismic module in SAPHIRE. This is intended to introduce users who are already familiar with seismic and PRA analyses to the SAPHIRE seismic module interface, including how to input data and perform seismic calculations. This is not intended to teach users how to do seismic-PRA.

The discussion assumes the availability of an internal-events PRA. Specifically, random-failure composed system-models, accident sequence progression, and initiating events have all been defined and developed for the engineered system of interest (e.g., nuclear power plant). The seismic analysis is being factored into that engineered system, which is already well understood and comprehensively modeled. Therefore, functional vulnerabilities have been identified and the seismic analysis consists of converting to and adding in the seismic-induced failures.

TOPICS

Building Model

SAPHIRE provides the flexibility to construct seismic risk analysis models by either (or a combination) of two methods. First, seismic-specific event tree and fault tree models can be developed via the graphical interface. Second, SAPHIRE contains a provision for performing transformations in the form of Boolean identities (i.e., A=A+B, A=B, or A=A*B). This allows the user to build on an internal events analysis when developing a seismic model. More specifically, after site-specific seismic vulnerabilities have been identified (through plant walk-downs or some other site-specific review), they can be incorporated into an existing internal events analysis using a set of basic-event transformations that either replace or add the seismic failure events to the existing basic events.

Hazard Curve

The hazard curve is the representation of the range of possible earthquakes. It is commonly found in the form of a probability of exceedence curve, with the earthquake ground acceleration on the horizontal and the probability of exceeding that acceleration on the vertical axes. (One source for this information is NUREG-1488.) However, SAPHIRE utilizes this information in the form of a histogram (or more precisely, a discreet probability density distribution). Specifically, the information needs to be arraigned into a maximum of 20 ground acceleration bins with each one assigned a yearly frequency of occurrence.

To input this information into SAPHIRE, the user selects "Modify " from the SAPHIRE menu. Selecting "Histograms" from the sub-menu brings up a dialog box containing the list histograms. Click the right mouse button for the pop-up menu containing options to be executed. Selecting "Add" allows the user to actually create the histogram. Each bin (numbered 1-20) is associated with an event name (e.g., HAZARDEVENT01), which is how that specific earthquake event (magnitude and frequency) is identified in the analysis. Note that these event names are generic. That is, only a single set of event names, corresponding to the 20 bins of a single histogram, are defined. The data associated with a particular event name is taken from the histogram that is selected for use with a particular family. This can cause confusion for the user when working with a histogram; ~~pressing <F1>~~ might list the event name corresponding to a particular bin, but the data might be from a totally different histogram. Which histogram will be used in the analysis is identified/changed by selecting the "Modify " - "Family" - "Modify" series of menu commands. Only the histogram listed in the "Medium" field, under the heading "Site Hazard Curves," is actually used. The "High" and "Low" fields are not used at this time. Uncertainty information for the earthquake data is entered directly into the basic event dialogs [i.e., from the histogram dialog, select the bin of interest and select "Modify " - "Basic Events" - event name (histogram bin name) of interest].

Event Trees

The most straightforward approach (at least with respect to using SAPHIRE) for creating a seismic analysis model involves the development of a seismic event tree that prioritizes and links the seismic-induced internal events initiators with the earthquake (the true initiating event). This single seismic event tree begins with a generic seismic-initiating event set to a value of 1.0. [The actual magnitude (g-level) and frequency of the earthquake of interest are identified by the user and factored into the analysis when the cut sets are generated and quantified.] The event tree top-events are those internal events initiators that have the potential to be induced by an earthquake. They are listed in order of severity (in terms of challenging plant safety systems), with the more severe induced-initiators listed first. This addresses the potential pitfall of over-counting core damage sequences (i.e., a single earthquake inducing both a large LOCA and a small LOCA at the same time). However, these event tree top-events are treated as seismic basic-events (or fault trees) with associated seismic fragility data. The resulting end states are therefore the frequencies of seismically induced challenges (i.e., internal-events type initiators) to the plant. These in turn can be identified as transfers to the systems analysis (internal-events accident sequences) event trees. This linking will automatically replace the internal events analysis initiator on the systems analysis event tree with the transferred information from the seismic event tree. This is how the linking between the earthquake, induced initiating event, and the system models are made.

Fault Trees - Building fault tree or converting random internal events fault tree

The actual system models are commonly fault trees. As mentioned above, the seismic system models can be created as independent, stand-alone fault trees. However, using SAPHIRE, they can also be integrated with the internal events analysis.

Along with linking (i.e., establishing transfers) between the seismic event tree and the accident sequence event trees, the system models supporting development of the accident sequence event tree top-events need be modified to include seismic-induced failures. This is done by defining transformations of the random failures modeled in the internal events analysis into seismic failures. In effect, these transformations execute Boolean identities of the original random-failure events, equating them with one or more seismic-induced failure events in the system logic model. The transformations can be performed such that the original event is kept in the model and the seismic event is simply added. This allows the user the option of incorporating random failures in the seismic analysis.

The transformations are created in the "Modify " menu selection. However, before creating a transformation, the seismic basic events must be added to the SAPHIRE database (see below). Once an event is identified as being susceptible to seismic initiators (on the "Modify Event" dialog, item 4 under "Susceptibilities"), SAPHIRE will automatically look for transformations whenever a seismic analysis is performed.

Basic Event Data - Fragilities and Uncertainty Data

Basic events are defined in the basic event database (select "Modify " then "Basic Events"). Before transformations can be created, the seismic basic events must be defined. Seismic failure data is usually characterized by a median fragility and two uncertainty terms representing the random uncertainty and the confidence uncertainty (Beta-R and Beta-U, respectively). There is also an added factor that might or might not be included in seismic failure data called the structural response factor (SRF). The SRF quantifies the amount of amplification or dampening of ground motion a particular piece of equipment experiences during an earthquake, by virtue of its location. For example, during a postulated earthquake, a relay on the fourth floor of a building would likely experience a different magnitude of shaking compared to a relay on the first floor. The SRF accounts for this difference. SAPHIRE does not maintain the SRF information separately. Before entering the seismic fragility data, the SRF needs to be factored in, then the SRF-adjusted fragility data is entered into the database.

To enter seismic data into a seismic basic event record, go to the "Failure Data Calculation Type" and click on the down arrow. Selecting "G" or "H" defines the basic event as a seismic basic event. The "G" and "H" simply identify the basis for the assumed magnitude of the peak ground acceleration (PGA) or g-level, for initially generating cut sets .

Analysis

Generating Cut Sets - Screening G-Level

SAPHIRE requires a failure probability for each basic event to generate cut sets. For the seismic basic events, this requires an assumed earthquake magnitude to combine with the basic event fragility data to produce a failure probability. The user can choose from two options, which are set individually for each basic event. When basic event data are input, from the "Modify Event" dialog, the user selects the "Calculation Type." To define an event as a seismic event, this field must be set to either the "G" or "H" option. The selected option determines how screening values are calculated for the seismic basic events. The "G" allows the user to input an assumed g-level for use in initially generating cut sets directly in the basic event information. The "H" specifies the use of the highest g-level (bin) from the histogram identified for use with that particular family (database) in SAPHIRE. Note that if the user expects to enable the cut set truncation feature when generating cut sets, the most prudent approach is to specify the highest g-level (i.e., PGA in units of gravity) that might be considered in subsequent calculations. Otherwise, cut sets that might be important in later analyses would be truncated because their seismic-induced failure probabilities fall below the truncation limit.

Quantifying Cut Sets

Seismic cut set quantification is performed similar to a normal, internal events (i.e., "Analysis Type = random") quantification, with a couple of minor differences. First, the "Analysis Type" needs to be set to seismic. This tells SAPHIRE to use the seismic cut sets. Second, after selecting "Quantification," the user is prompted to choose the "G-Level" for which the quantification is to be performed. The options available include: each g-level bin that contains non-zero data for the hazard histogram identified for use with the current Family, all bins together, and all bins separately. Once the g-level is selected, the quantification proceeds and results are calculated.

Results

An important feature to keep in mind is that only a single cut set list is maintained for each system, sequence, or end state in SAPHIRE. This limit also applies to seismic calculation, which is where the effect of this can cause some confusion. Specifically, when performing a seismic quantification, the cut set lists for each g-level are not maintained. Hence, the user is limited when viewing and reporting quantified cut sets; only the last quantification performed (i.e., that specific g-level) will be available. Numerical results, however, are stored and available for each individual g-level.

Set (Defn.)

A *set* is a collection of objects or elements with some characteristics or distinguishing features in common. An example of a set is all possible states of the components in a nuclear power plant.

Population (Defn.)

The set of all elements is called the *population*, the *reference set*, the *universal set*, or the *identity set*.

The population is denoted by Ω or by I.

Null Set (Defn.)

The set not containing any elements is called the *null set*, the *empty set*, and sometimes the *zero set*.

The null set is denoted by ∅.

Subset (Defn.)

Let *A* and *B* be sets of Ω in the following discussion. *B* is said to be a *subset* of *A*, if and only if every element in *B* is also an element of *A*.

***B is a subset of A* is denoted by B $\subseteq$ A.**

Proper Subset (Defn.)

If *A* contains an element not in *B*, then *B* is called a *proper subset* of *A*.

***B is a proper subset of A* is denoted by B $\subset$ A.**

Set Equality (Defn.)

A and *B* are said to be *equal*, denoted by *A*=*B*, if *A* and *B* have the same elements and neither is a proper subset of the other.

A = B if and only if A $\subseteq$ B and B $\subseteq$ A.

Constrained Noninformative Distribution

The constrained noninformative prior distribution is a diffuse distribution with the specified mean. The amount of diffuseness is set so that the distribution is noninformative except for the information given by the specified mean, neither too concentrated nor too diffuse. The precise definition, although not needed by SAPHIRE users, is given by Atwood (1994): The parameter of interest is an initiating event frequency or a basic event probability. Transform the model so that the parameter of interest is approximately a location parameter. In this transformed model, find the maximum entropy distribution constrained by the specified mean for the original parameter. This distribution is as flat as possible subject to the constraint. Then transform back to the original model. The resulting distribution is the constrained noninformative distribution on the parameter.

As originally developed, the distribution is a Bayes prior distribution, intended to be updated based on data. In that context, when the unknown parameter is a frequency and the data are Poisson counts, the constrained noninformative prior has the form of a gamma distribution. When the unknown parameter is a probability and the data are binomial counts, the constrained noninformative prior can be approximated well by a beta distribution.

Therefore, SAPHIRE uses the gamma form of the constrained noninformative prior for quantifying uncertainty in a frequency, λ, and the beta form of the constrained noninformative prior distribution for quantifying uncertainty in a probability, *p*. The user specifies only the mean. SAPHIRE then decides on the form of the distribution and calculates the other parameter to achieve the right diffuseness.

Preliminary Screening of Distributions

Whenever a distribution is specified, for either the frequency of an initiating event or the probability of a basic event, SAPHIRE performs some preliminary checks, to help the user avoid entering illegal or unwise values.

Screening for Illegal Parameter Values

The parameters of the distribution are always checked for legality. For example, the mean must be nonnegative and the other parameter(s) must be nonnegative or positive, for all of the distributions listed in Table 2. When a violation is found, SAPHIRE prints a message on the screen stating what is wrong. The user may not leave the screen until legal parameters have been entered for the specified distribution. When a parameter is legal but degenerate, such as a zero standard deviation or a zero range, SAPHIRE prints a warning message, but allows the value to be used.

Screening for Range

In this discussion, the term *parameter* refers not to the numbers that define a distribution (such as a mean and variance) but to the quantity that has the uncertainty distribution, namely, the frequency of an initiating event or the probability of a basic event. SAPHIRE looks at the probability that the uncertain parameter is outside of its allowed range. That is, the probability of a basic event must lie between 0 and 1, and the frequency of an initiating event must be greater than 0. These are the allowed ranges of the uncertain parameters. If, for example, the parameter is assigned a normal distribution, it is possible for the parameter to take a negative value, which is illegal. If, in addition, the parameter is a basic event probability, it must not exceed 1.0, but the normal distribution (and some others) allows this with positive probability. SAPHIRE prints a warning message on the screen if the probability of exceeding the maximum legal value is greater than 5E!4, and a second warning message if the probability of being less than the minimum legal value is greater than 5E!4. The user may then change the distribution or ignore the warnings.

Later, if an illegal value is generated in the course of Monte Carlo simulation, the value is discarded and a new value is generated. If Latin Hypercube Sampling is performed instead of simple Monte Carlo simulation, the range is restricted to the legal portion of the distribution. Thus, in either case the distribution is truncated to its legal portion. When the truncated portion has a very small probability, the effect of the truncation is negligible. When a substantial fraction of original distribution is illegal, however, the effect of the truncation is to change the distribution, including its mean, from what the user specified. Therefore, it is unwise to use a distribution that puts substantial probability outside the allowed range of the parameter.

Screening for Variance

Whenever a distribution is specified, SAPHIRE compares it to the constrained noninformative distribution having the same mean. As described in Section 9.3.9, the constrained noninformative prior is a diffuse distribution with the specified mean, and with the amount of diffuseness corresponding to ignorance of everything except the mean. If the user specifies a distribution having a variance larger than that of the constrained noninformative distribution, SAPHIRE prints a warning message on the screen. The warning states that the specified distribution is very diffuse, even more diffuse than a model of ignorance. The user has put more weight in the tails of the distribution than the constrained noninformative distribution does, and so the user may be overly pessimistic.

There can be valid reasons for choosing a distribution that is more diffuse than the constrained noninformative distribution. For example, the mean, treated as known for the constrained noninformative distribution, may not be known very well, and the user may desire to model this uncertainty by adding some extra spread to the distribution. Or the user may be duplicating portions of published earlier work, where a highly diffuse distribution was used. If, however, the user simply picked a distribution with a large variance, in an attempt to model great uncertainty, it may be preferable to use the constrained noninformative distribution or one with a comparable variance.